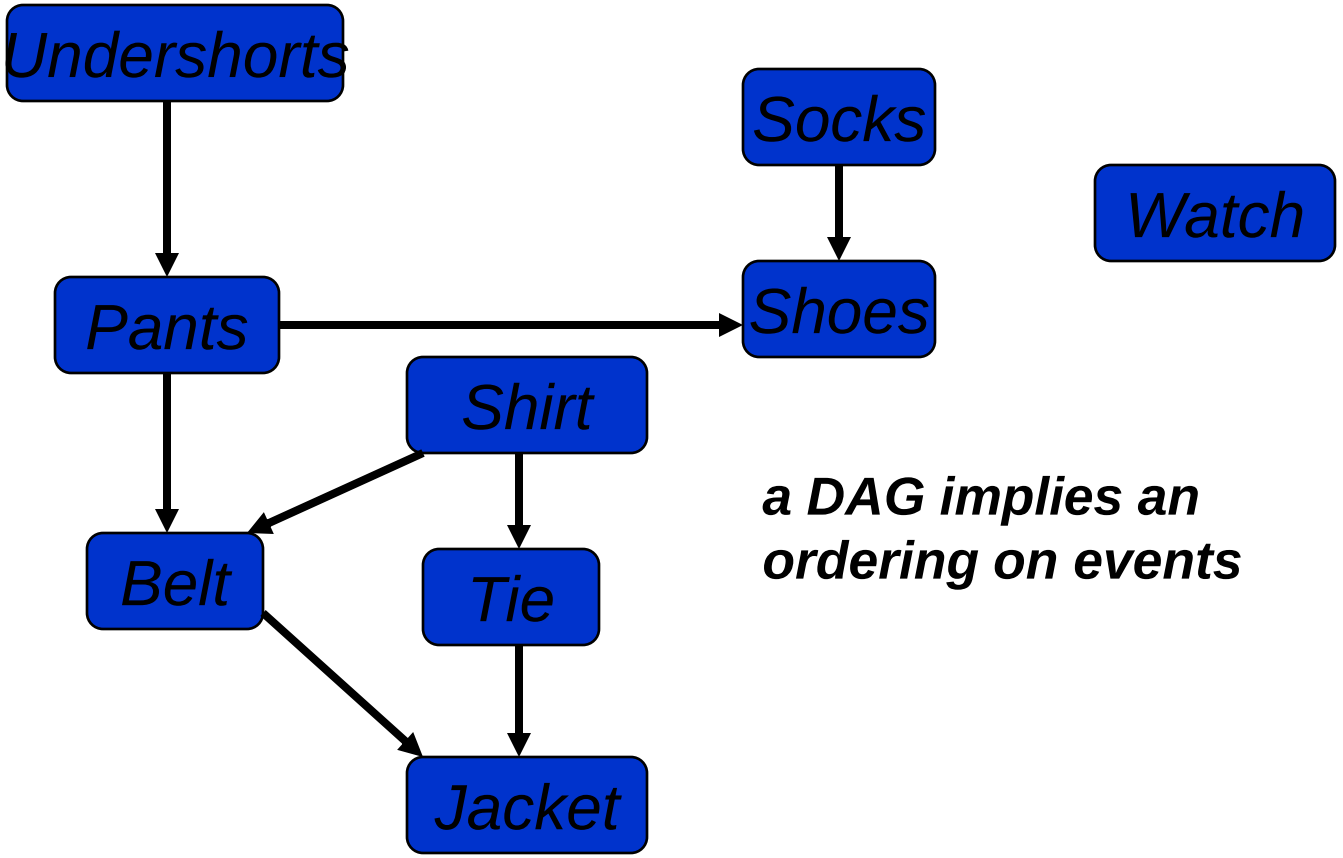
Algorithms

Topological Sort

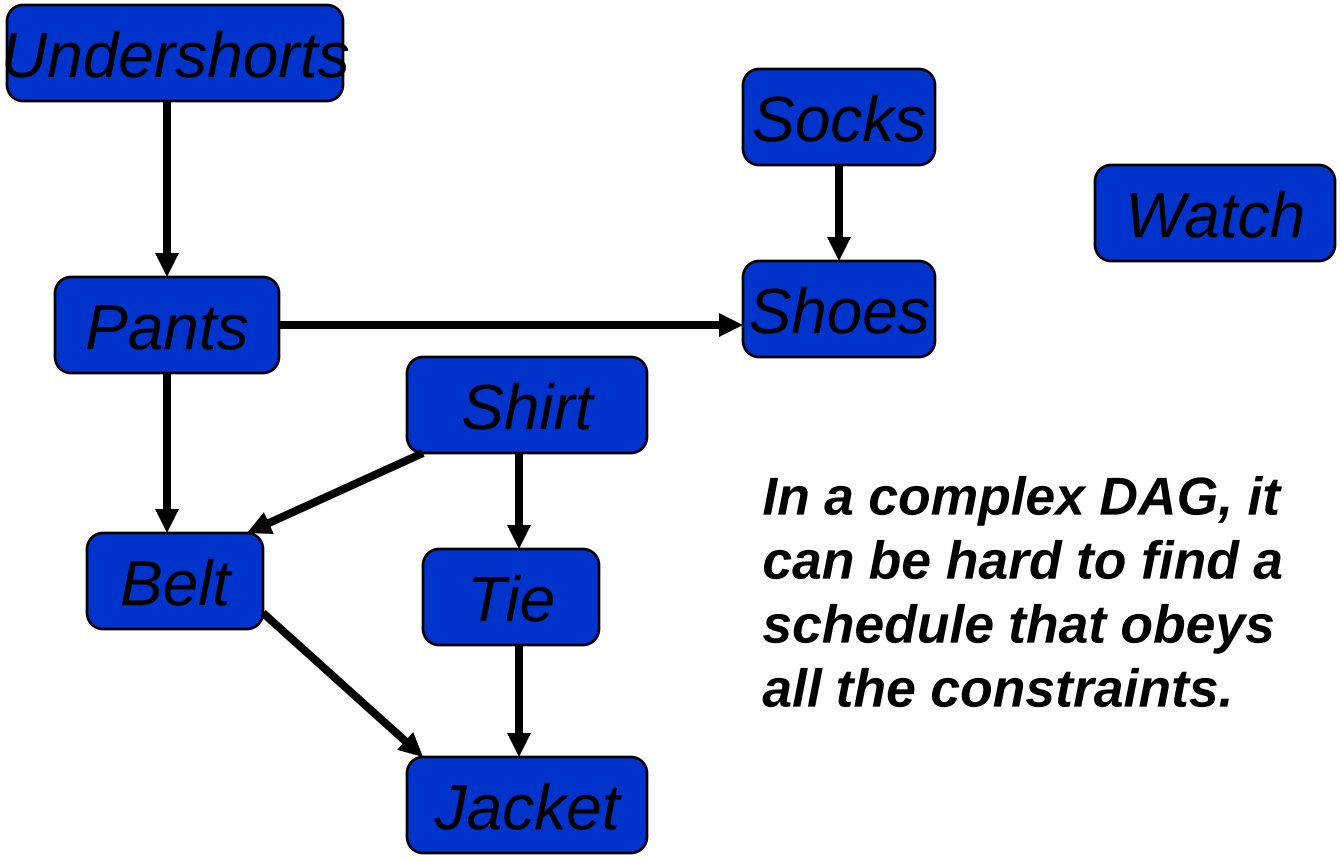
Strongly connected components
Minimum Spanning Trees

Example DAG



a DAG implies an ordering on events

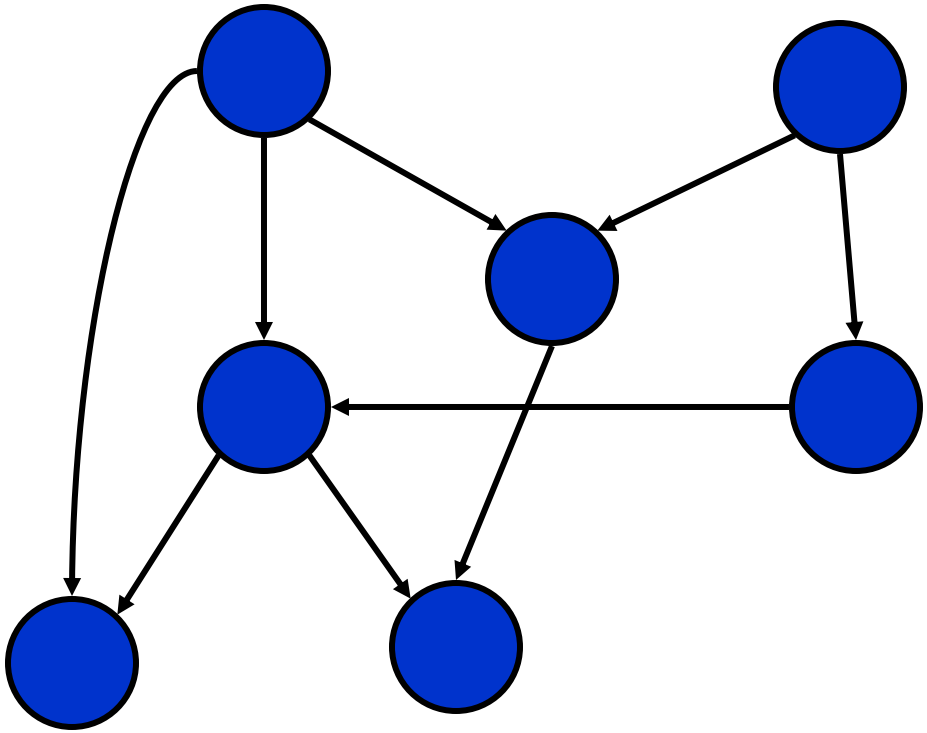
Example DAG

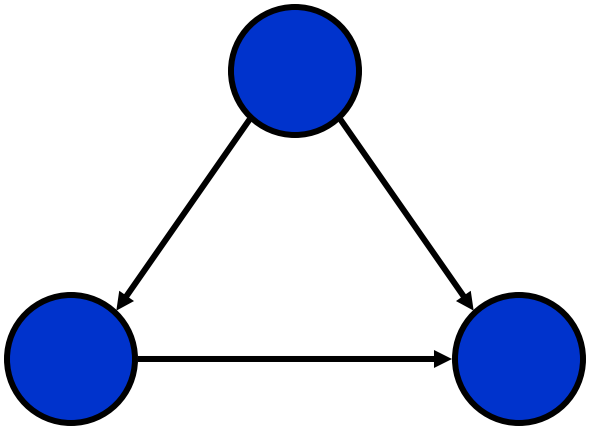


In a complex DAG, it can be hard to find a schedule that obeys all the constraints.

Directed Acyclic Graphs

- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles:





DFS and DAGs

- Theorem: a directed graph G is acyclic iff a DFS of G yields no back edges:
  - => if G is acyclic, will be no back edges
    - Trivial: a back edge implies a cycle
  - <= if no back edges, G is acyclic
    - Proof by contradiction: G has a cycle $\Rightarrow \exists$ a back edge
      - Let *v* be the vertex on the cycle first discovered, and *u* be the predecessor of *v* on the cycle
      - When *v* discovered, whole cycle is white
      - Must visit everything reachable from *v* before returning from DFS-Visit()
      - So path from u$\rightarrow$v is gray$\rightarrow$gray, thus (*u*, *v*) is a back edge

Topological Sort

Topological Sort

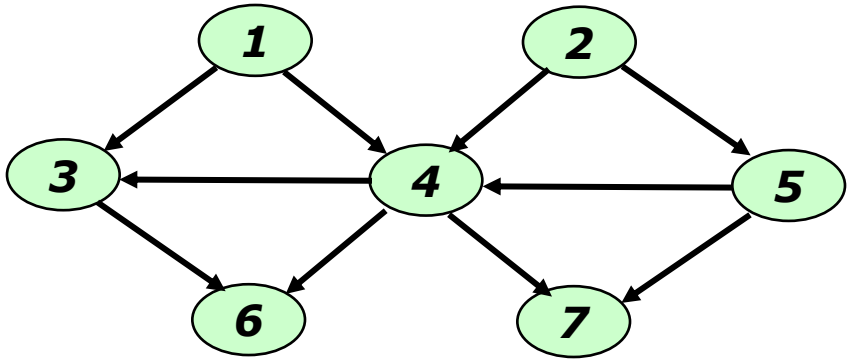
- For a directed acyclic graph G = (V,E)
- A topological sort is an ordering of all of G's vertices $v_1, v_2, \ldots, v_n$ such that...
- vertex *u* comes before vertex *v* if edge $(u, v) \in G$

Formally: for every edge (v_i, v_k) in *E*, i<k.

Visually: all arrows are pointing to the right

Topological sort

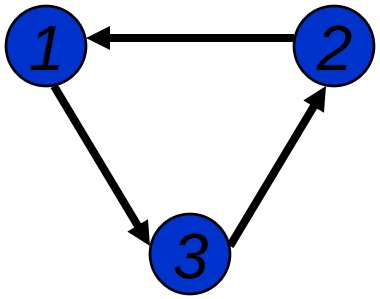
- There are often many possible topological sorts of a given DAG
- Topological orders for this DAG :
  - 1,2,5,4,3,6,7
  - 2,1,5,4,7,3,6
  - 2,5,1,4,7,3,6
  - *Etc.*



- Each topological order is a *feasible schedule*.

Topological Sorts for Cyclic Graphs?

Impossible!



- *If v and w are two vertices on a cycle, there exist paths from v to w and from w to v.*
- *Any ordering will contradict one of these paths*

A Topological Sort Algorithm

```
Topological-Sort()
{
   1. Call DFS to compute finish time f[v] for
      each vertex
   2. As each vertex is finished, insert it onto
      the front of a linked list
   3. Return the linked list of vertices
}
```

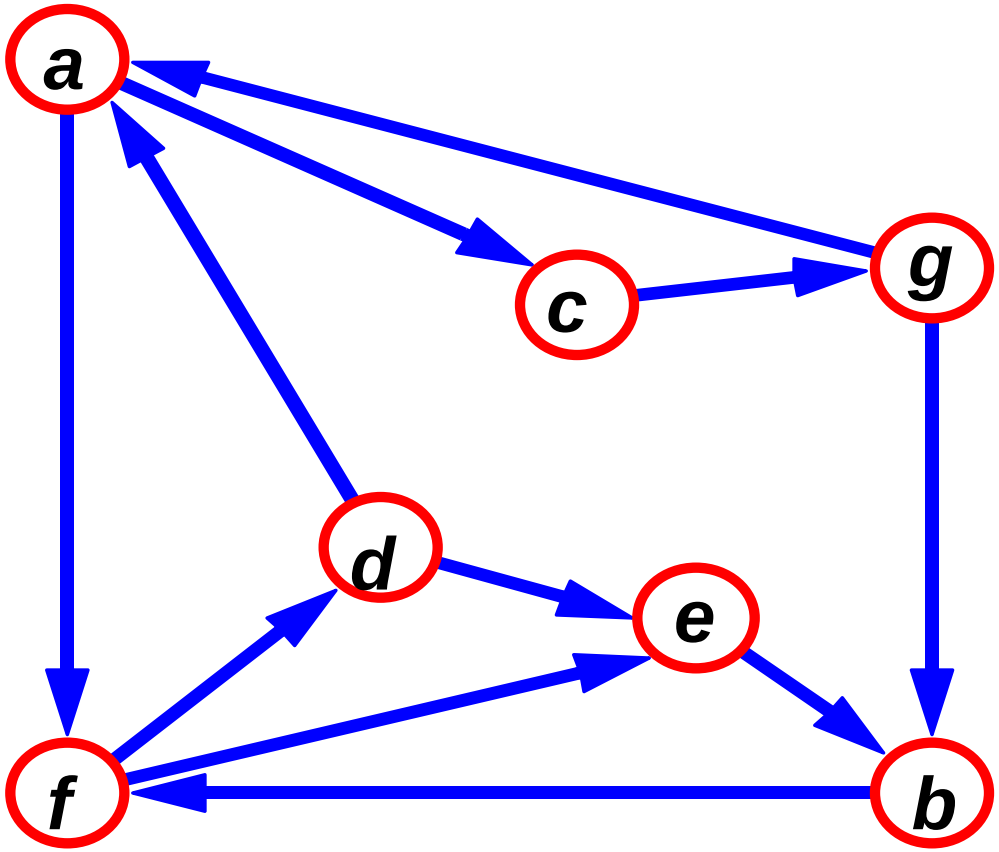
- Time: O(V+E)
- Correctness: need to prove that
  $(u,v) \in G \Rightarrow f[u]>f[v]$

Correctness of Topological Sort

- Lemma: $(u,v) \in G \Rightarrow f[u] > f[v]$
  - When (u,v) is explored, u is gray, consider the following cases:
    1. v is gray $\Rightarrow$ (u,v) is back edge. Can't happen, if G is a DAG.
    2. v if white $\Rightarrow$ v becomes descendent of u $\Rightarrow$ $f[v] < f[u]$
       (since must finish v before backtracking and finishing u)
    3. v is black $\Rightarrow$ v already finished $\Rightarrow$ $f[v] < f[u]$

Strongly Connected Directed graphs

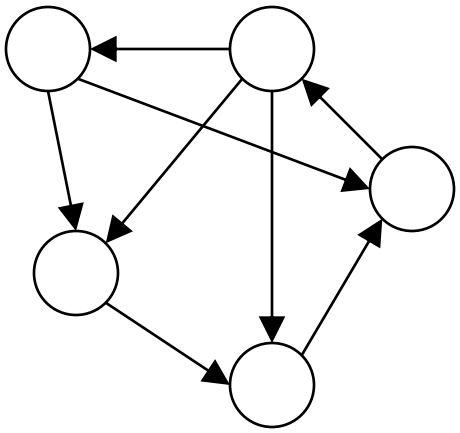
- Every pair of vertices are reachable from each other



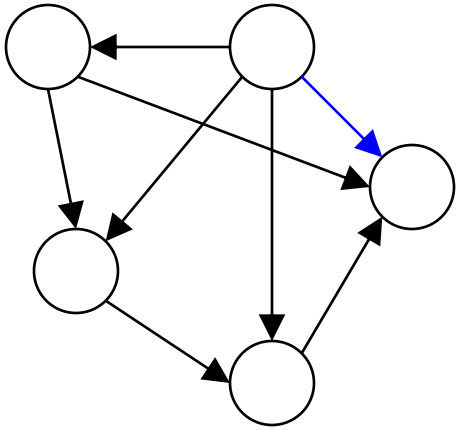
Strongly-Connected

Graph G is ***strongly connected*** if, for every u and v in V, there is some path from u to v and some path from v to u.

Strongly Connected



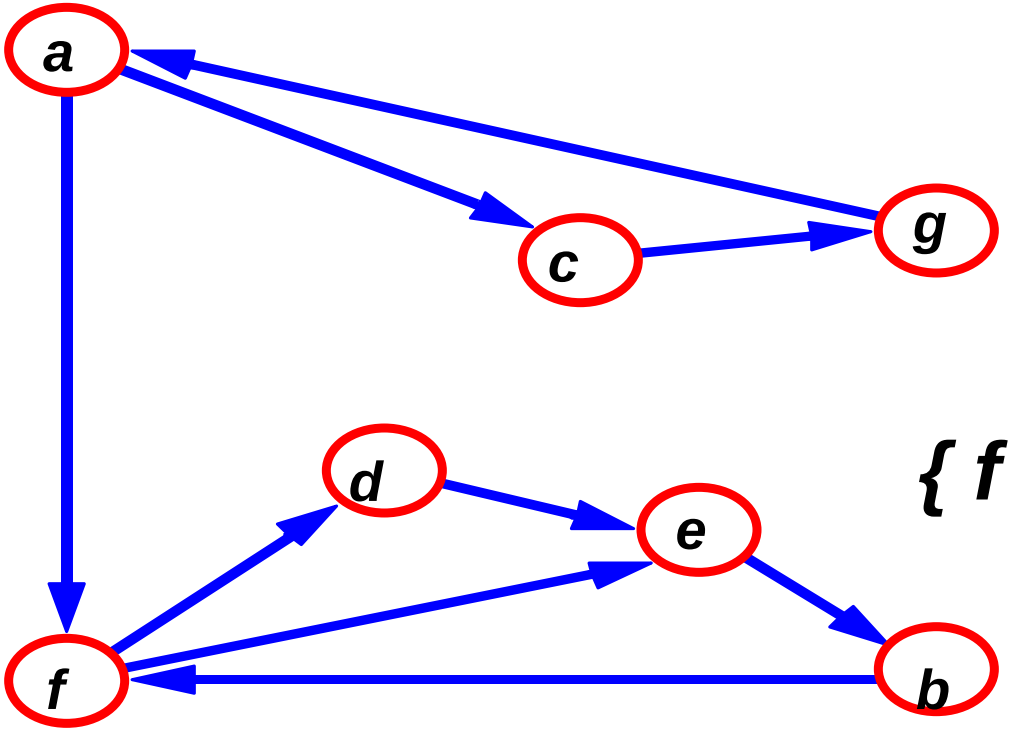
Not Strongly Connected



Strongly-Connected Components

A strongly connected ***component*** of a graph is a maximal subset of nodes (along with their associated edges) that is strongly connected. Nodes share a strongly connected component if they are **inter-reachable**.

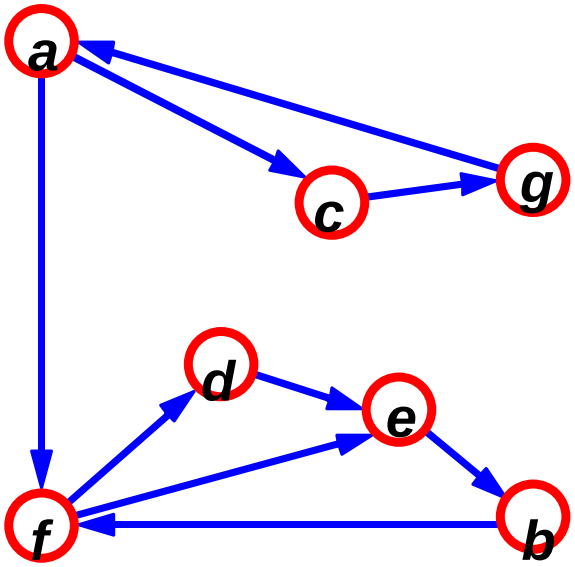
Strongly Connected Components

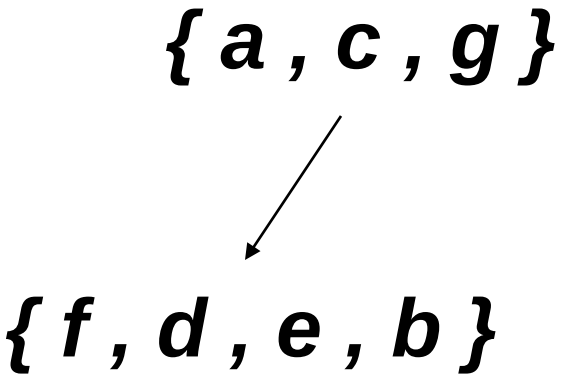


{ a , c , g }

{ f , d , e , b }

Reduced Component Graph of Strongly Connected Components





- Component graph $G^{SCC}=(V^{SCC}, E^{SCC})$: one vertex for each component
  - $(u, v) \in E^{SCC}$ if there exists at least one directed edge from the corresponding components

Strongly-Connected Components

Strongly-Connected-Components(G)

1. call DFS(G) to compute finishing times f[u] for each vertex u.
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f[u]
4. output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component.

The graph G^T is the transpose of G, which is visualized by reversing the arrows on the digraph.

Runtime

Lines 1 and 3 are $\Theta(E+V)$ due to DFS

Line 2 involves creating an adjacency list or matrix, and it is also $O(E+V)$

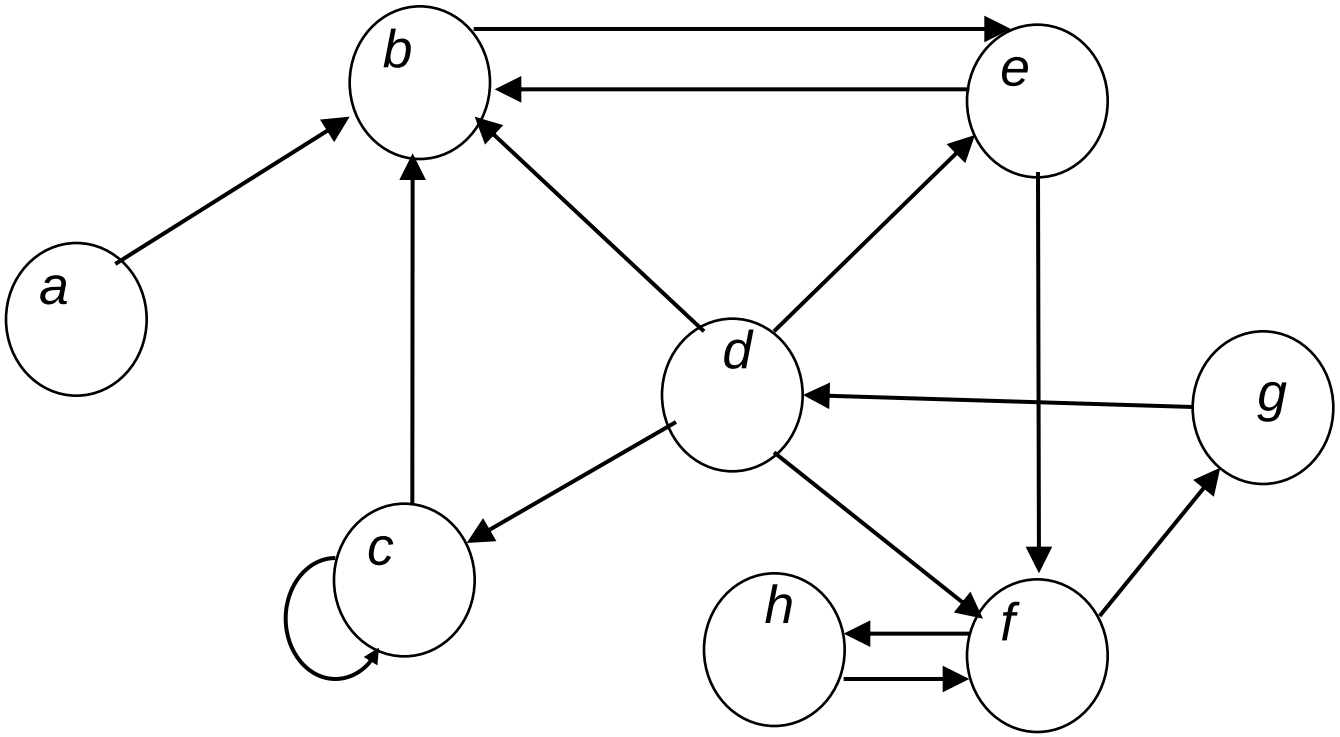
Line 4 is constant time

So, SCC(G) is $\Theta(E+V)$

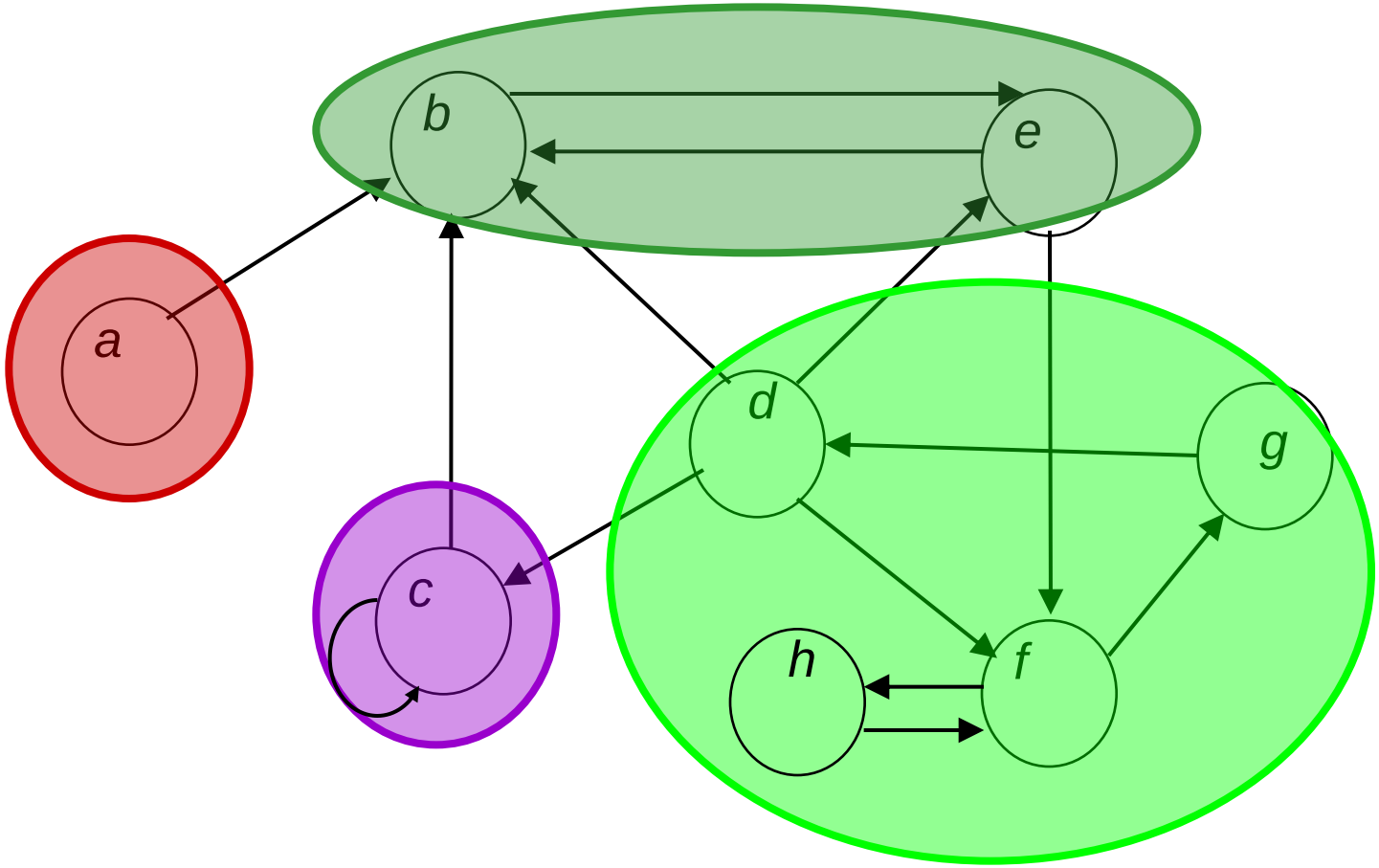
Strongly connected components

- Definition: the strongly connected components (SCC) $C_1, \ldots, C_k$ of a directed graph $G = (V,E)$ are the *largest* disjoint sub-graphs (no common vertices or edges) such that for any two vertices u and v in C_i, there is a path from u to v and from v to u.
- Equivalence classes of the binary $path(u,v)$ relation, denoted by $u \sim v$.
- Applications: networking, communications.
- Problem: compute the strongly connected components of G in linear time $\Theta(|V|+|E|)$.

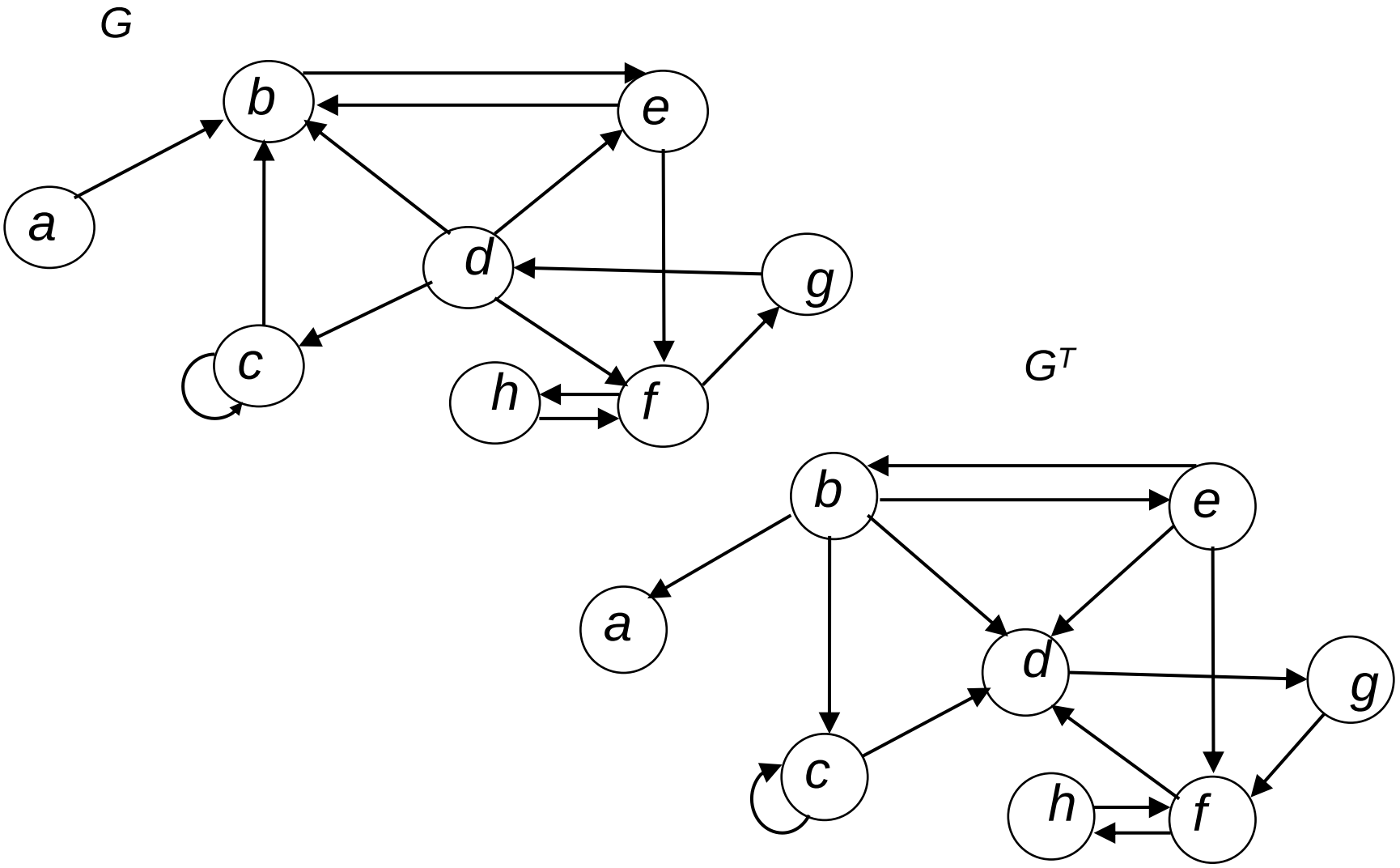
Example: strongly connected components



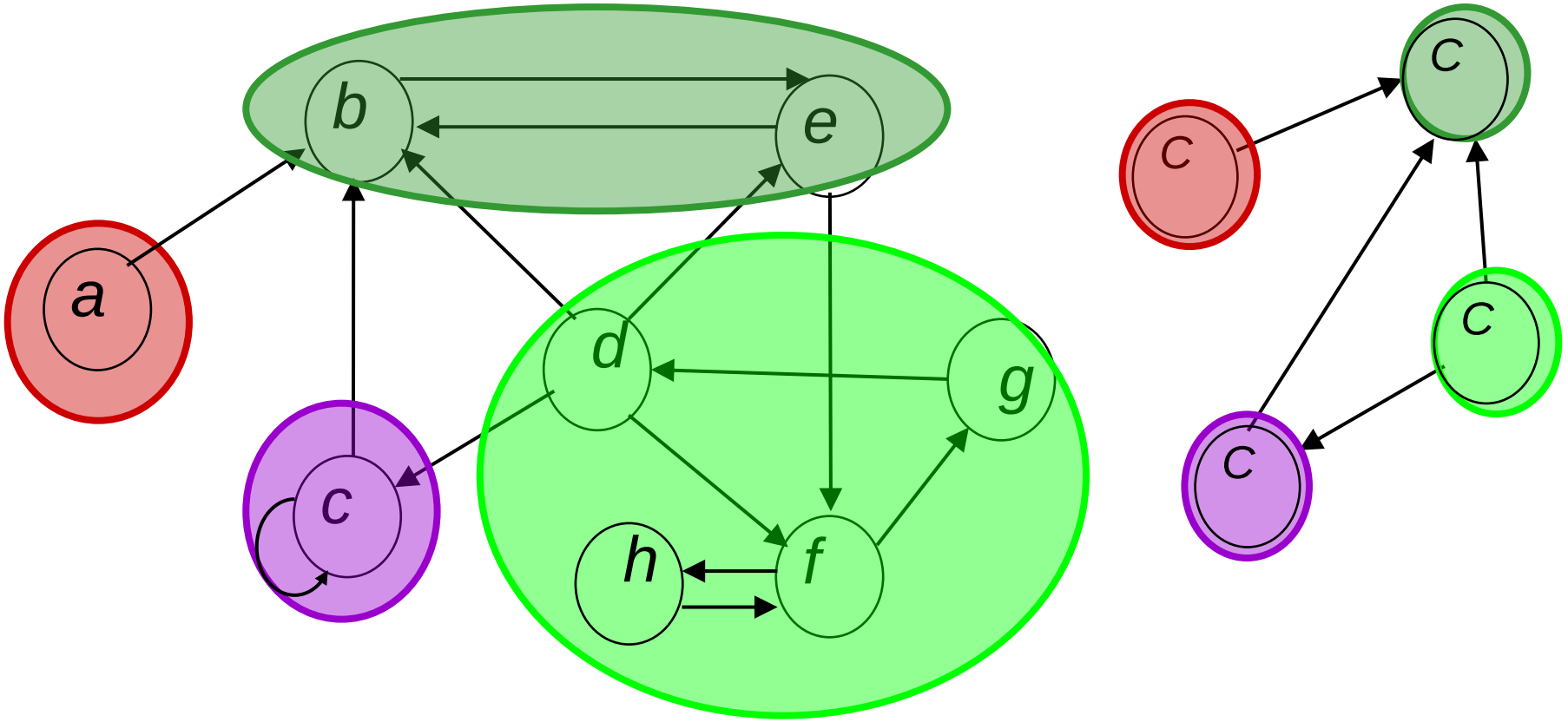
Example: strongly connected components



Example: transpose graph G^T

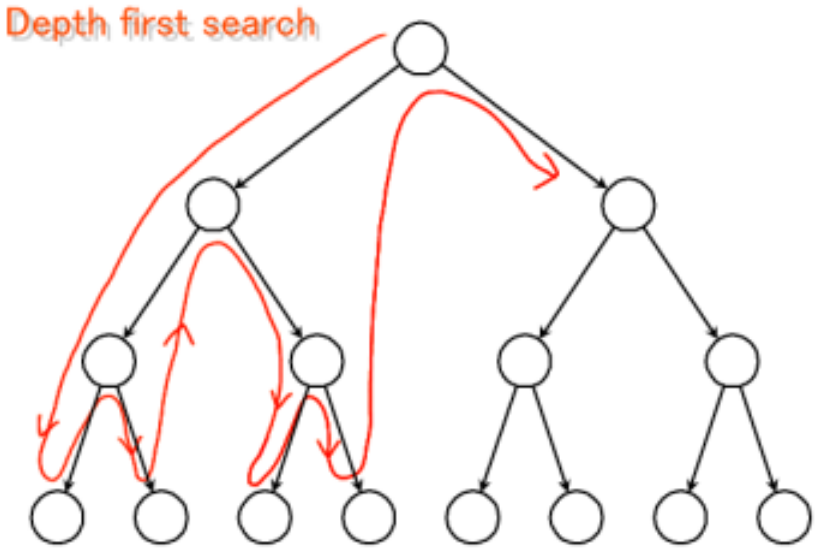


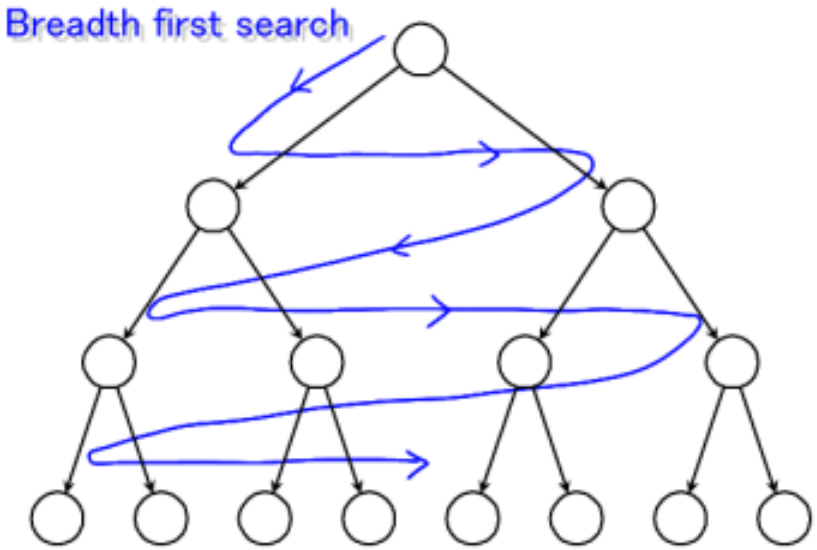
Example: SCC graph



Graph Traversals

Graph Traversals





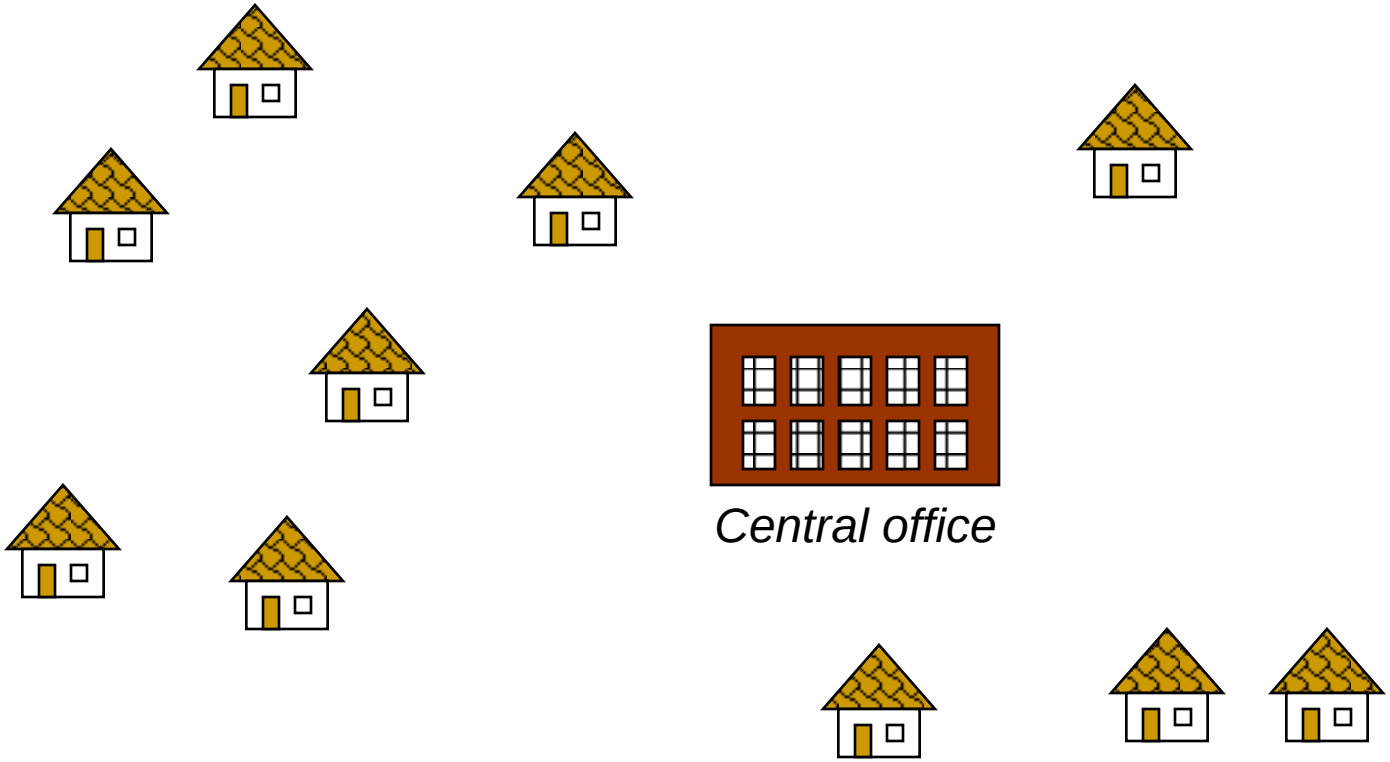
- *Both take time: O(V+E)*

Graph *Searching ???*

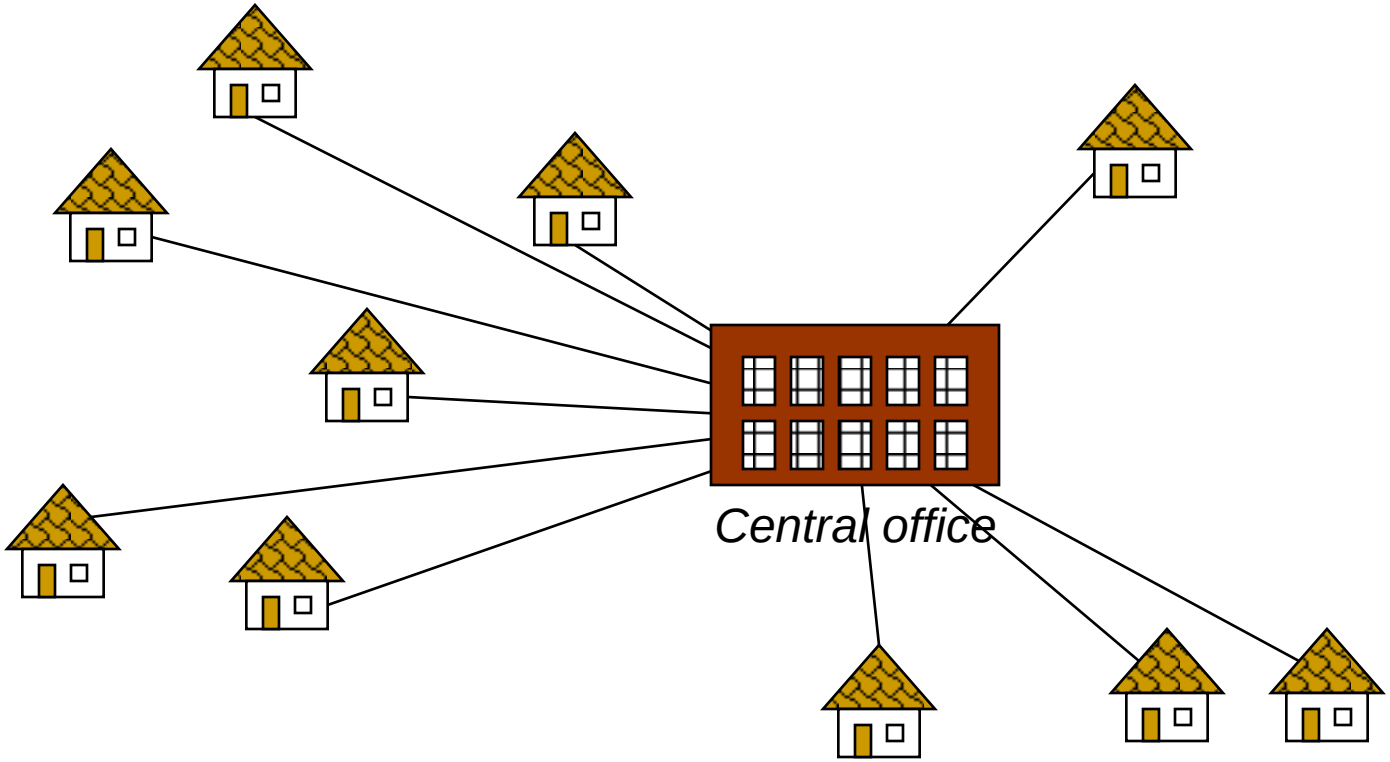
- Graph as state space (node = state, edge = action)
- For example, game trees, mazes, ...
- BFS and DFS each search the state space for a best move. If the search is exhaustive they will find the same solution, but if there is a time limit and the search space is large...
- DFS explores a few possible moves, looking at the effects far in the future
- BFS explores many solutions but only sees effects in the near future (often finds shorter solutions)

Minimum Spanning Trees

Problem: Laying Telephone Wire

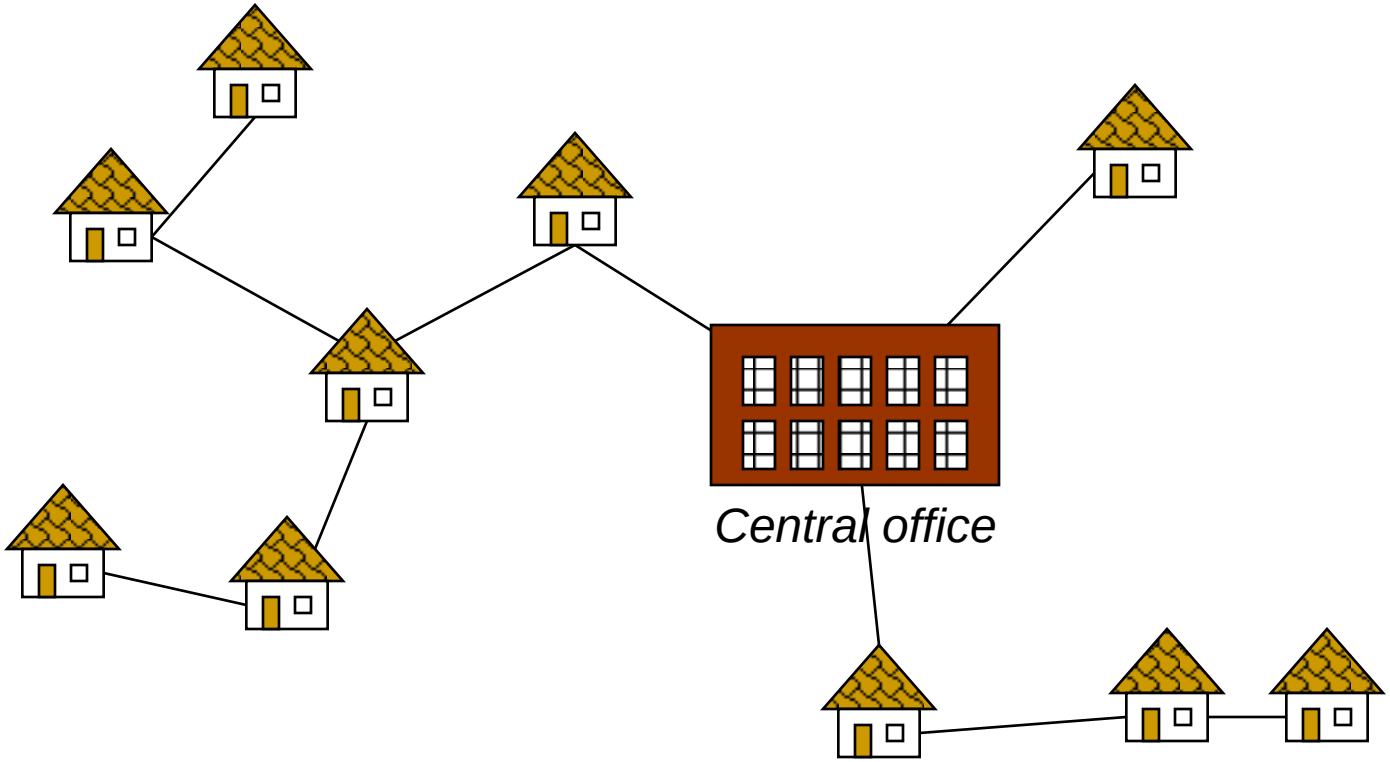


Wiring: Naïve Approach



Expensive!

Wiring: Better Approach



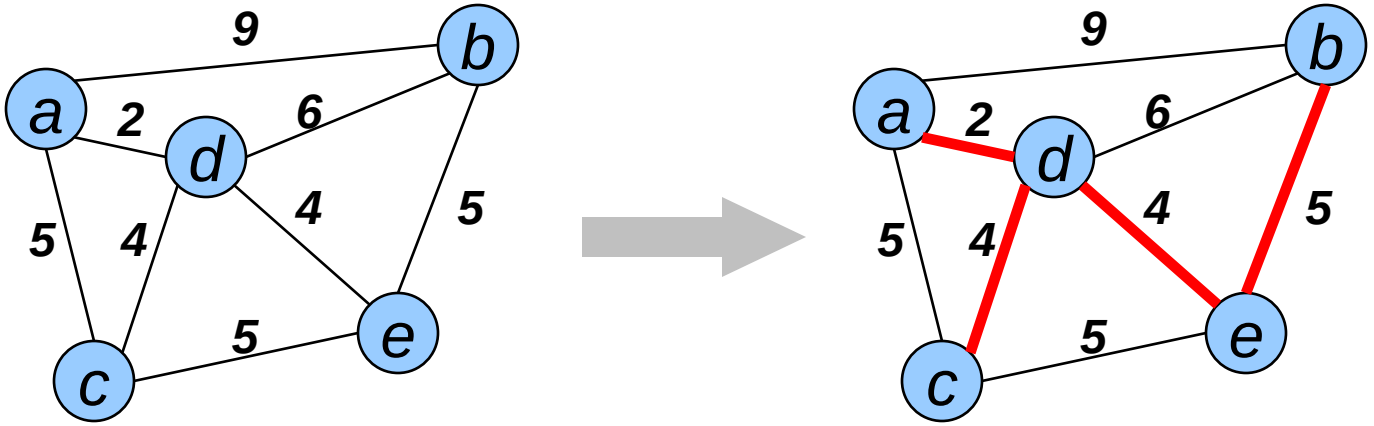
Minimize the total length of wire connecting the customers

Minimum Spanning Tree (MST)

*A **minimum spanning tree** is a subgraph of an undirected weighted graph **G**, such that*

- *it is a tree (i.e., it is acyclic)*
- *it covers all the vertices **V***
  - *contains **|V| - 1** edges*
- *the total cost associated with tree edges is the minimum among all possible spanning trees*
- *not necessarily unique*

How Can We Generate a MST?



Minimum Spanning Tree: Prim's Algorithm

- Prim's algorithm for finding an MST is a greedy algorithm.
- Start by selecting an arbitrary vertex, include it into the current MST.
- Grow the current MST by inserting into it the vertex closest to one of the vertices already in current MST.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

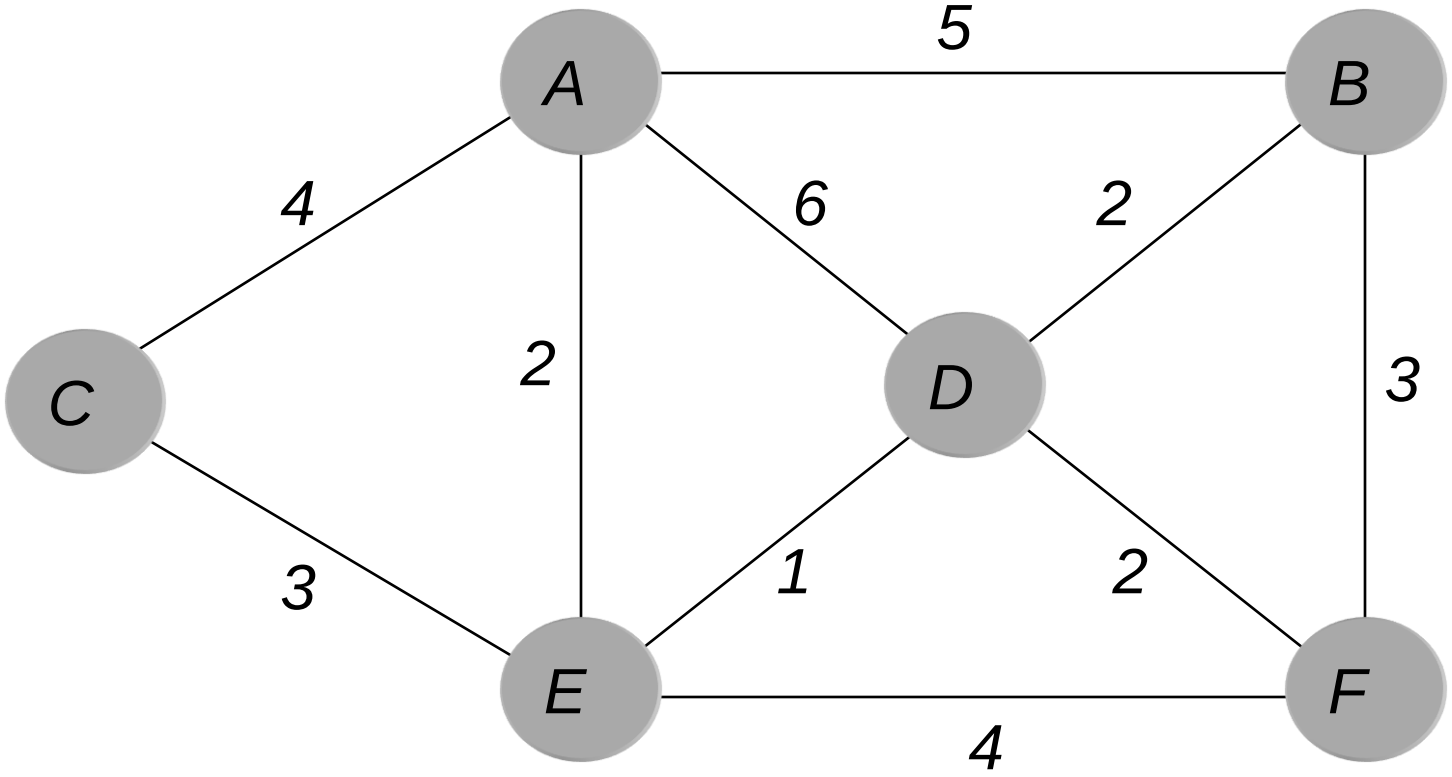
Prim‘s Algorithm

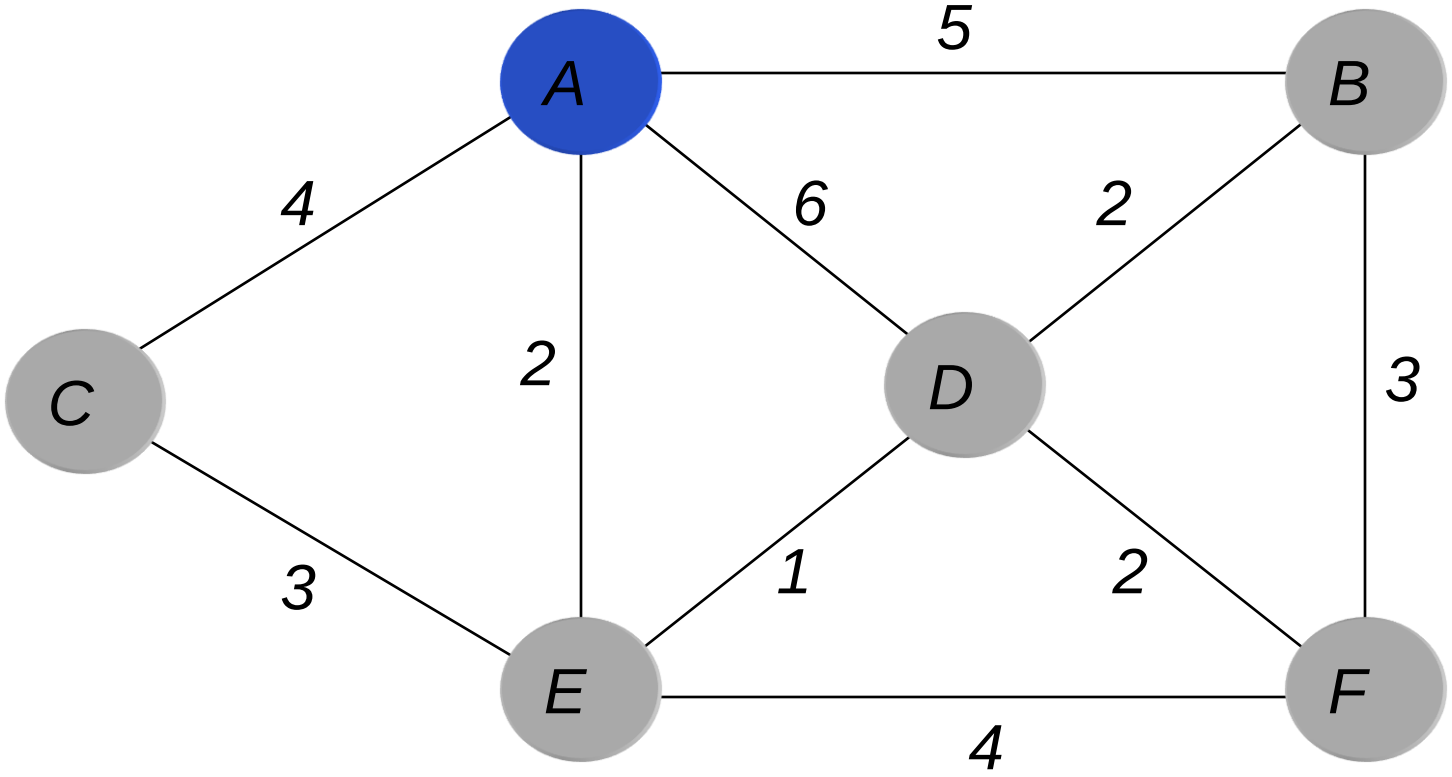
1. *All vertices are marked as not visited*

2. *Any vertex v you like is chosen as starting vertex and is marked as visited (define a cluster C)*

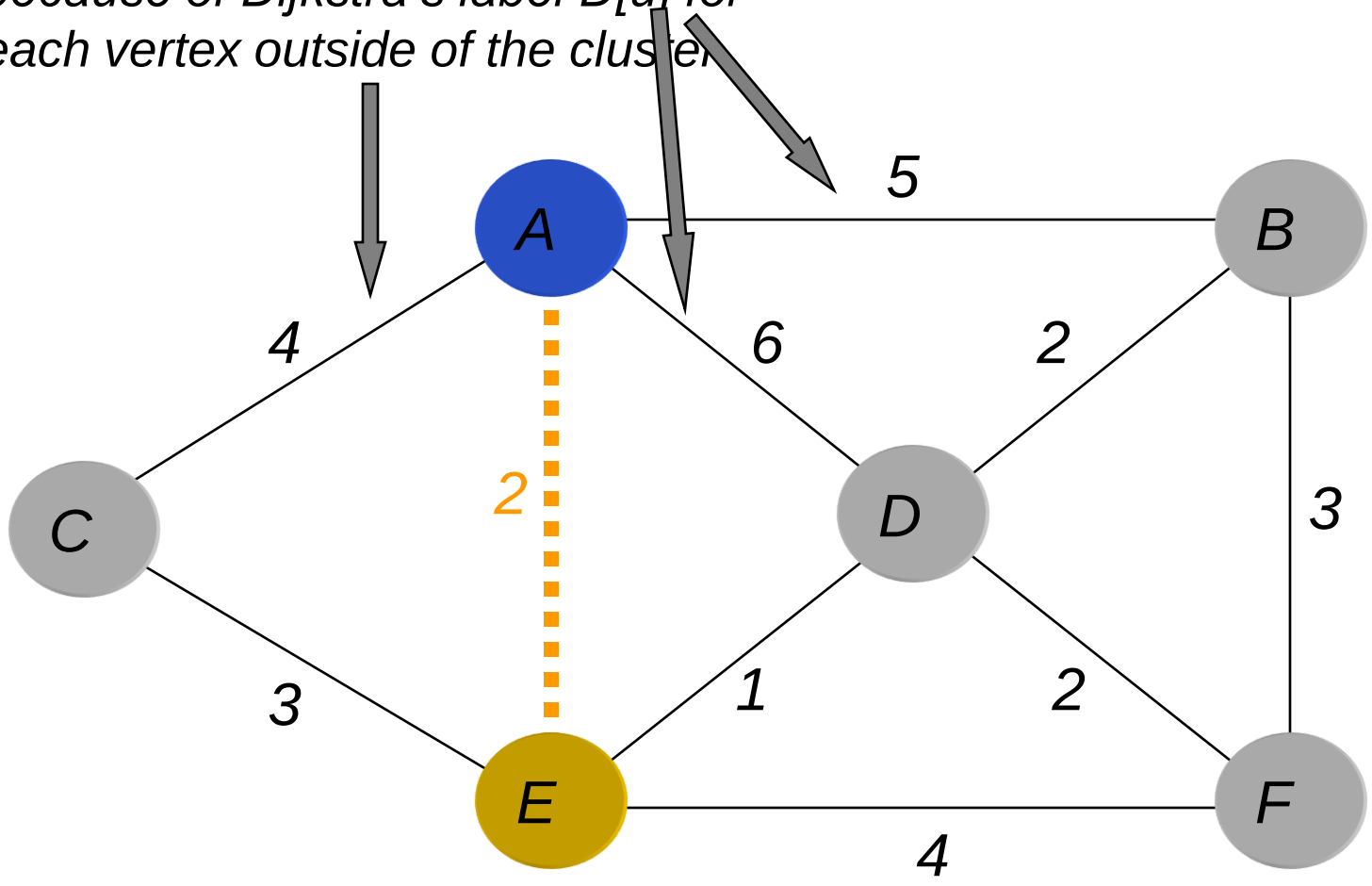
3. *The smallest- weighted edge e = (v,u), which connects one vertex v inside the cluster C with another vertex u outside of C, is chosen and is added to the MST.*

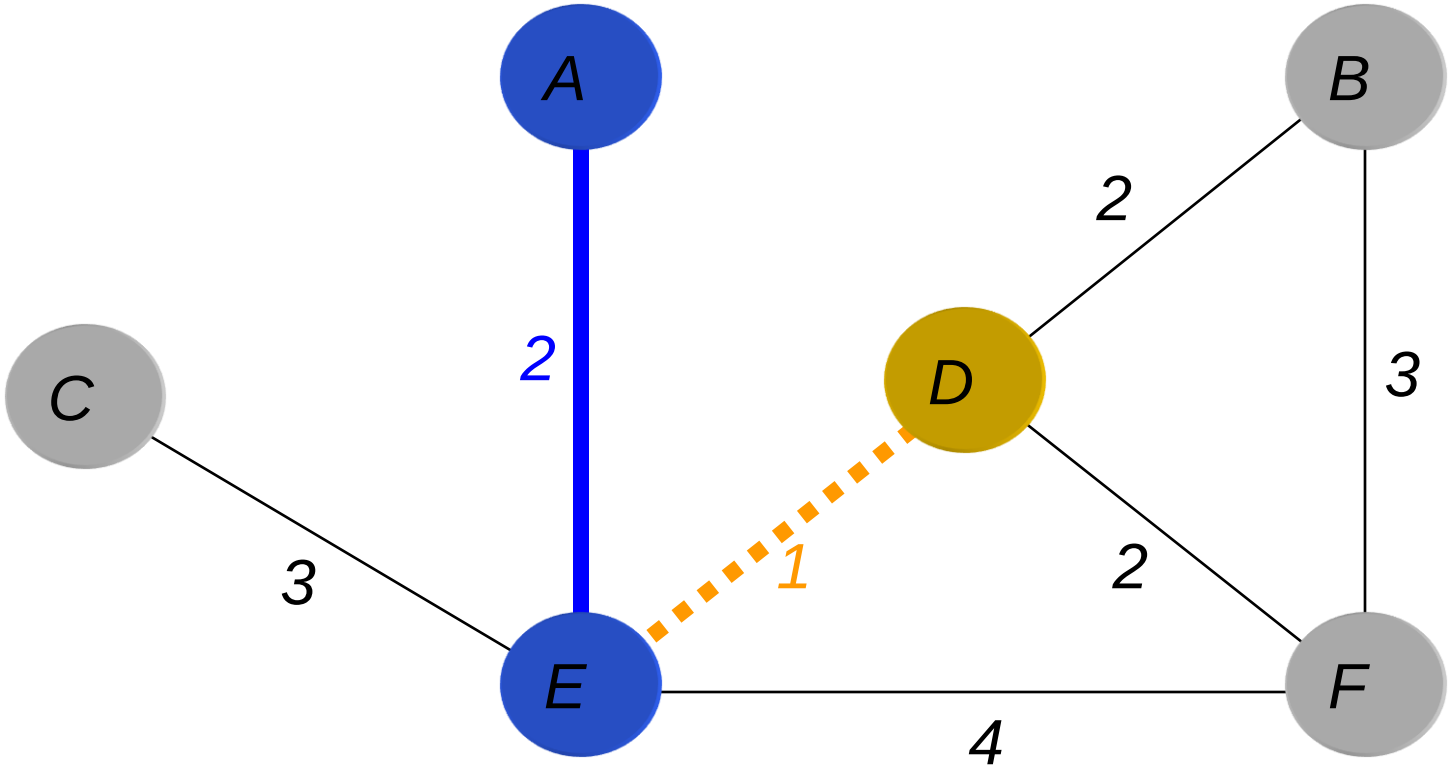
4. *The process is repeated until a spanning tree is formed*

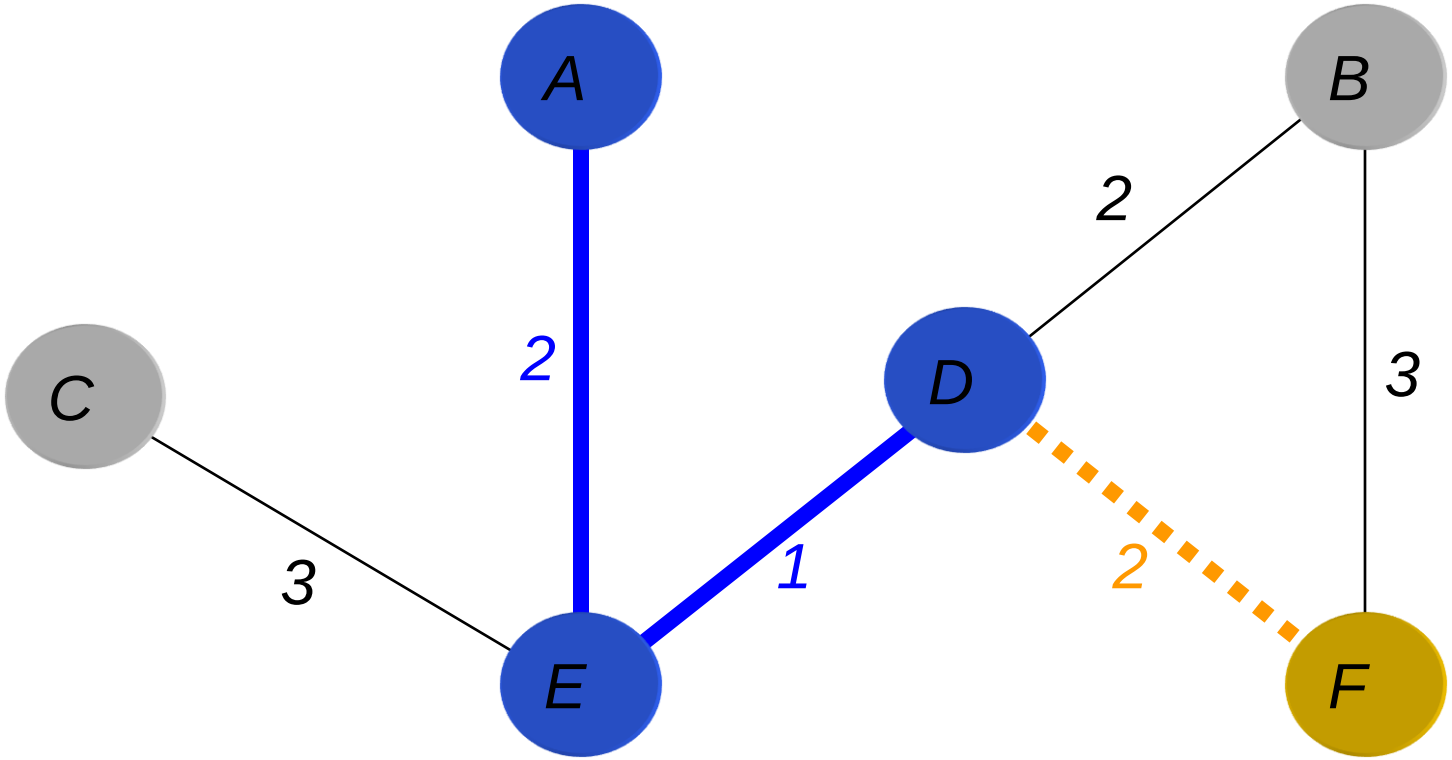

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

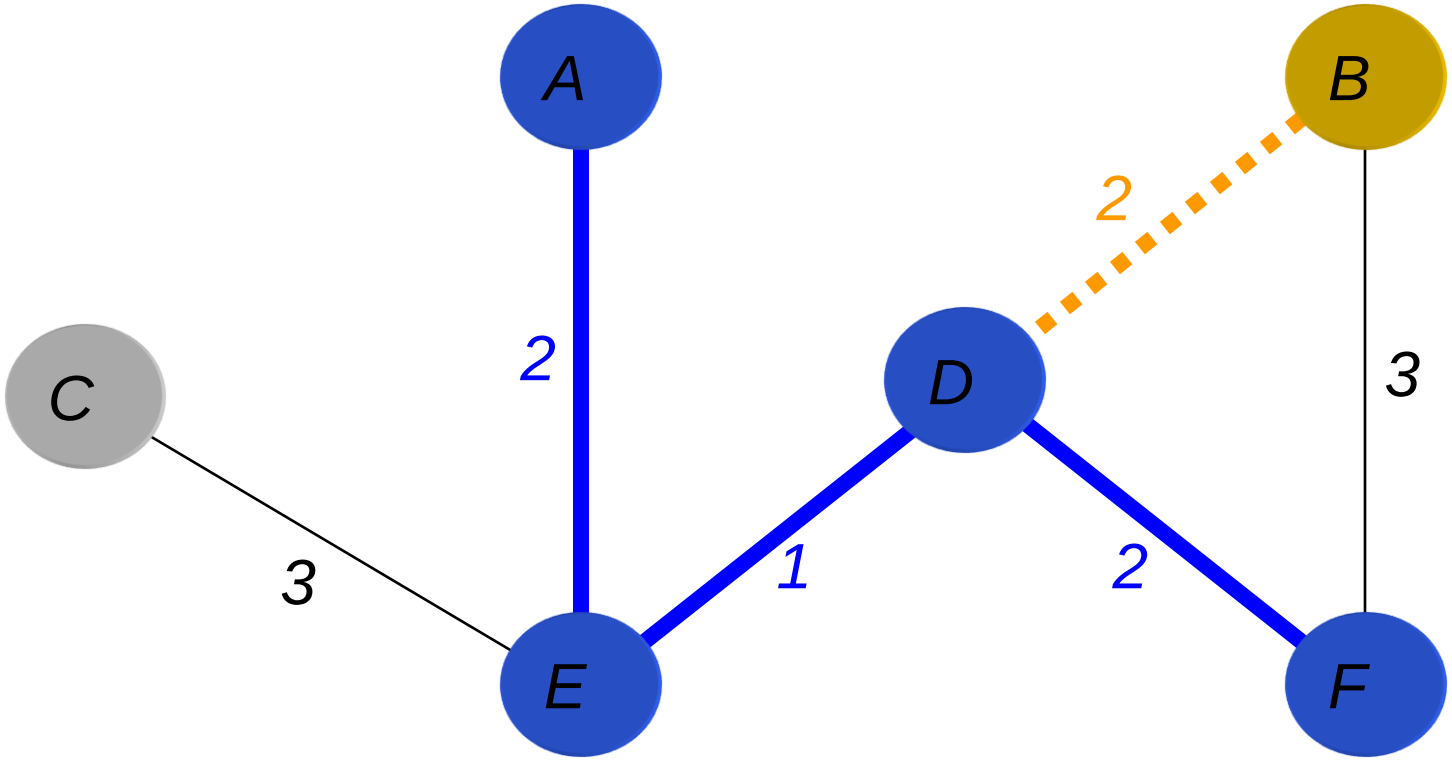

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

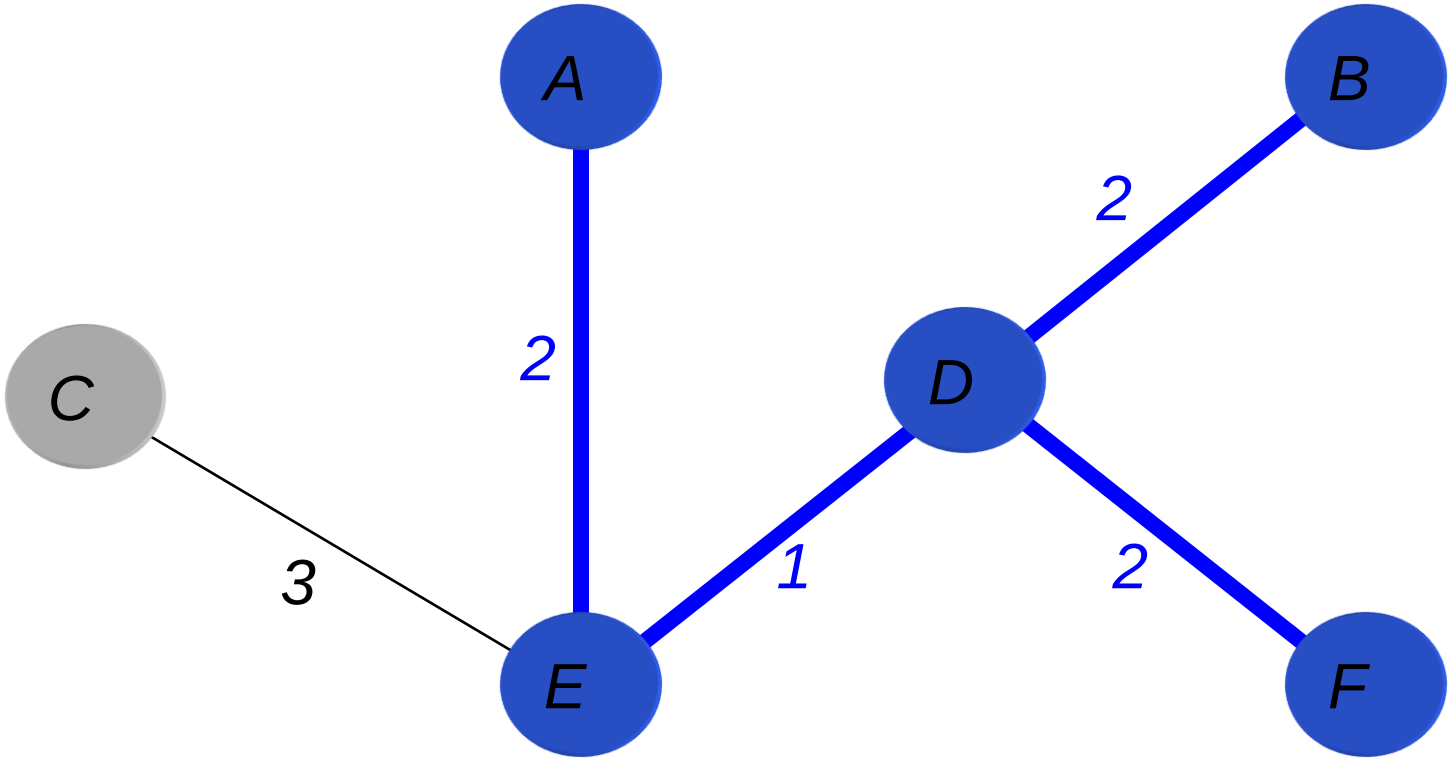
We could delete these edges because of Dijkstra's label D[u] for each vertex outside of the cluster

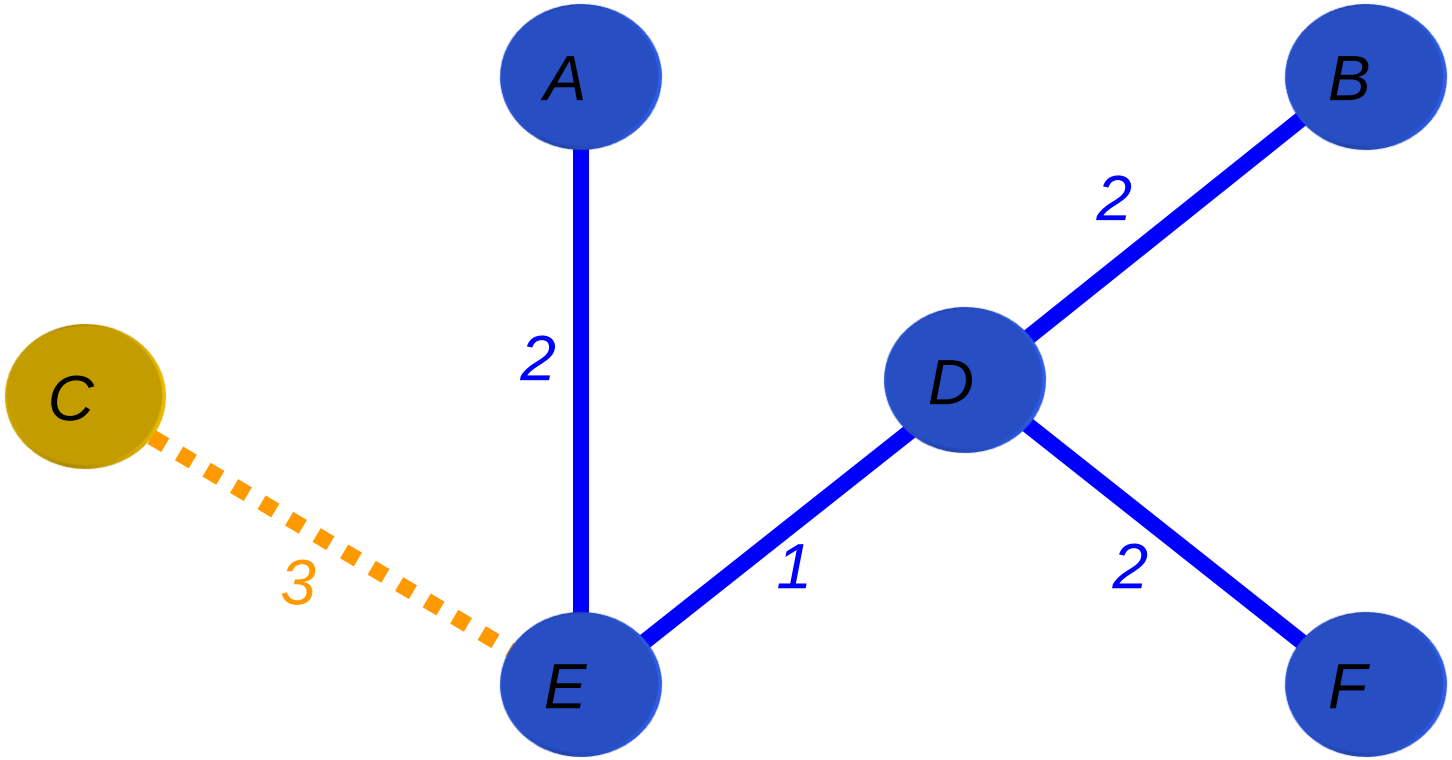



A
B
C
D
E
F
2
2
3
3
1
2
4

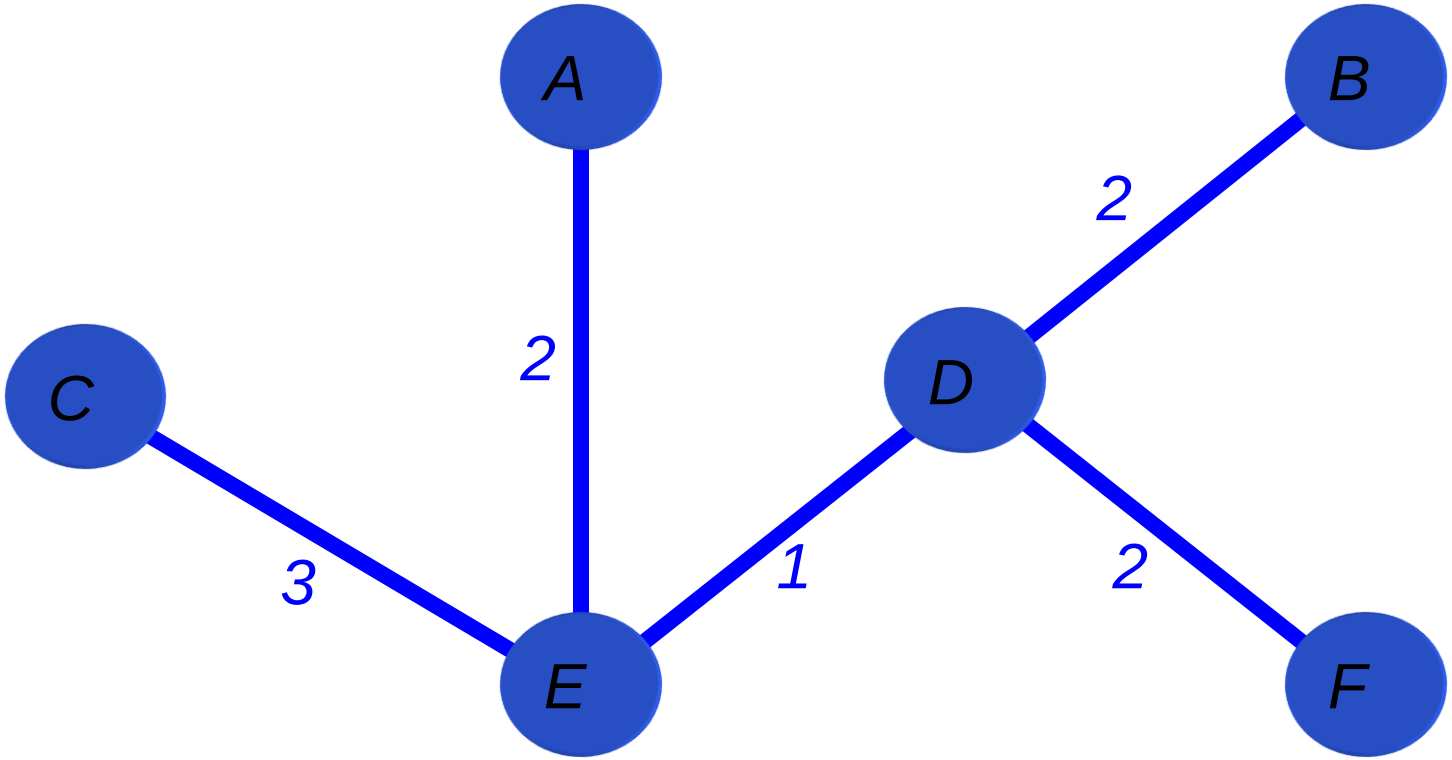

A
B
C
D
E
F
2
2
3
3
1
2

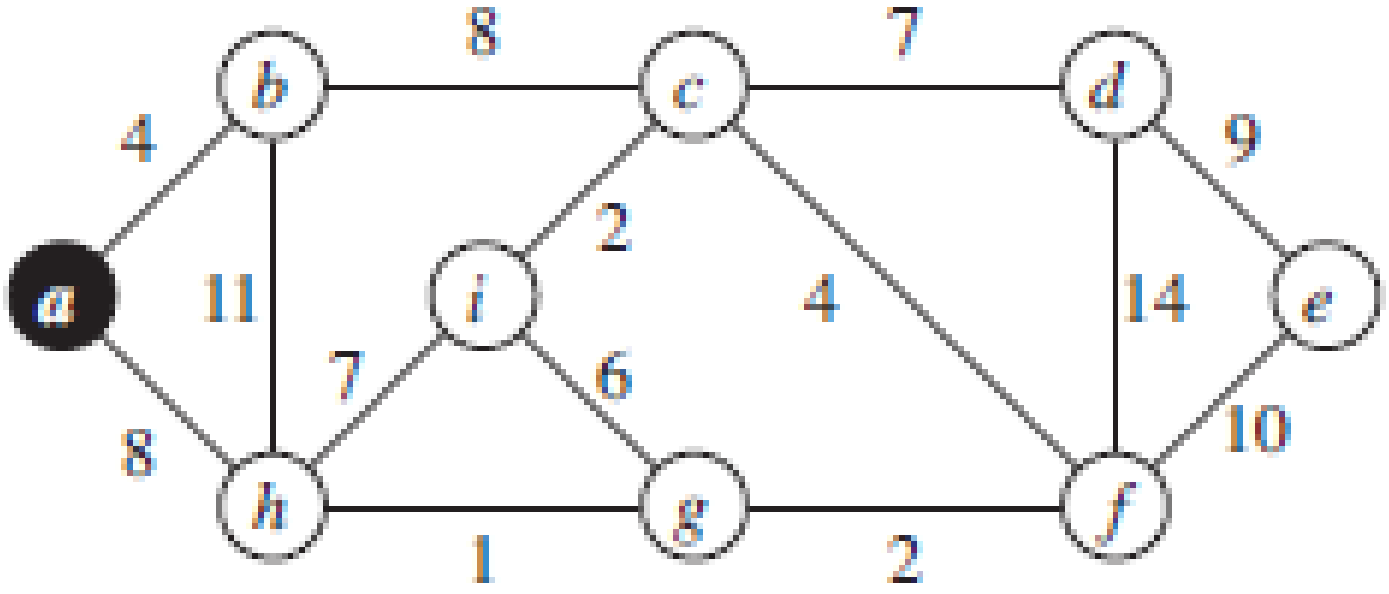

A
B
C
D
E
F
2
2
3
3
1
2

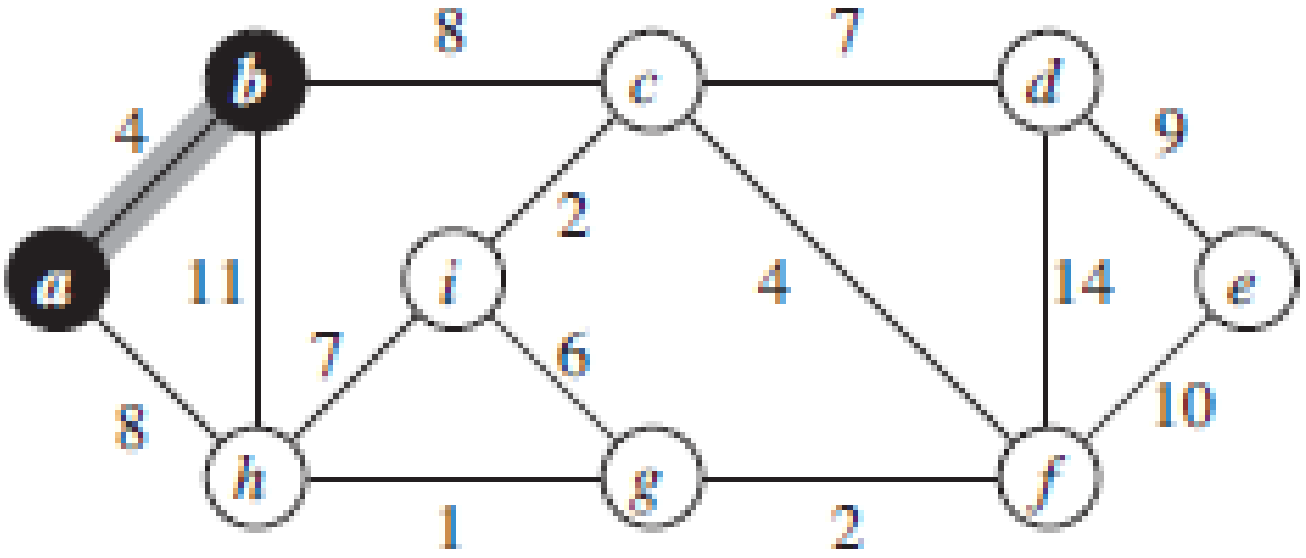

A
B
C
D
E
F
2
2
3
1
2

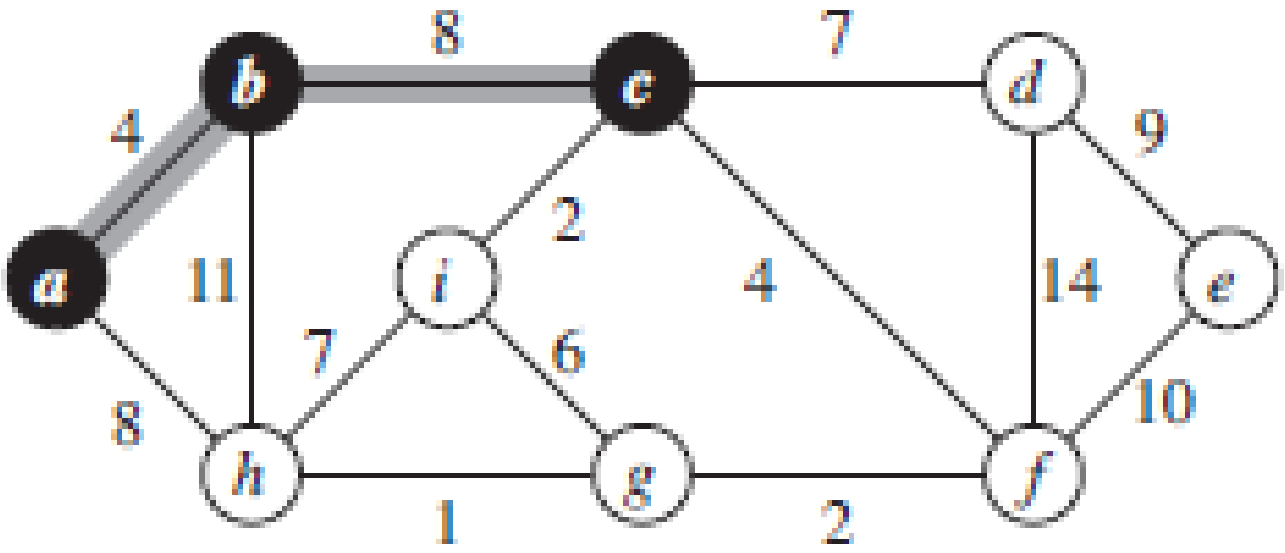

A
B
C
D
E
F
2
2
3
1
2

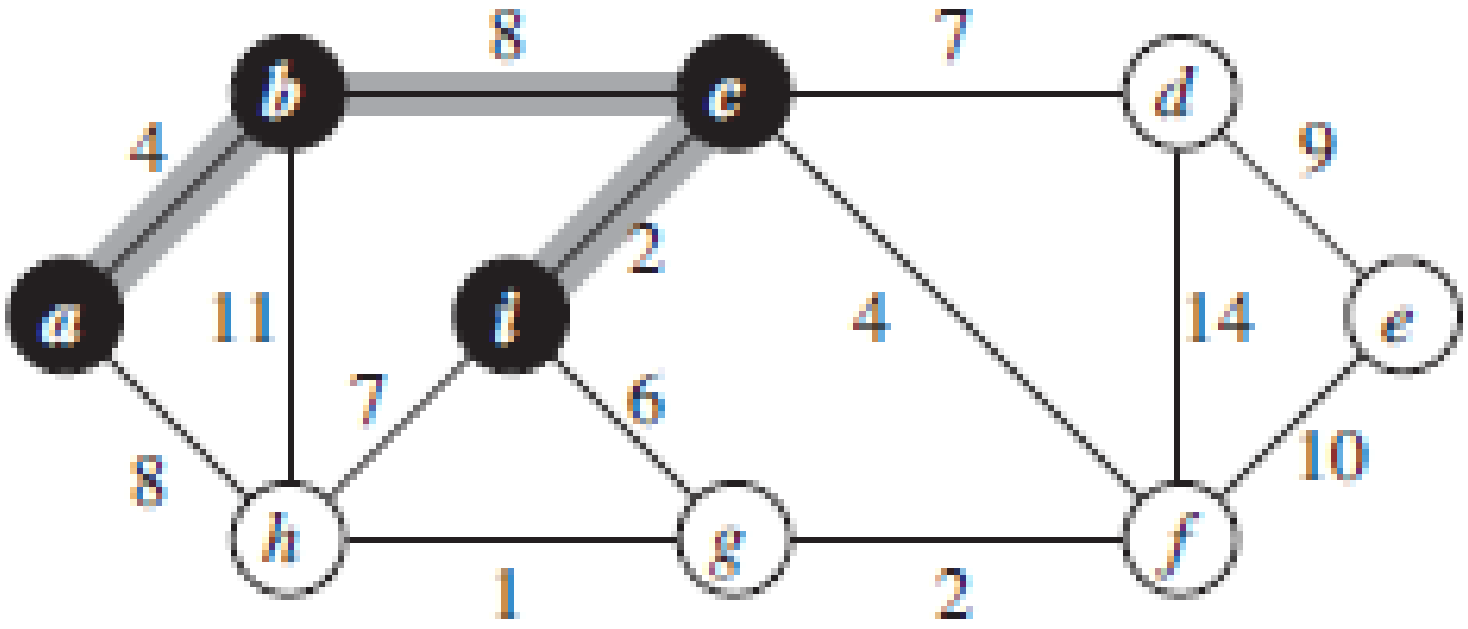
minimum- spanning tree

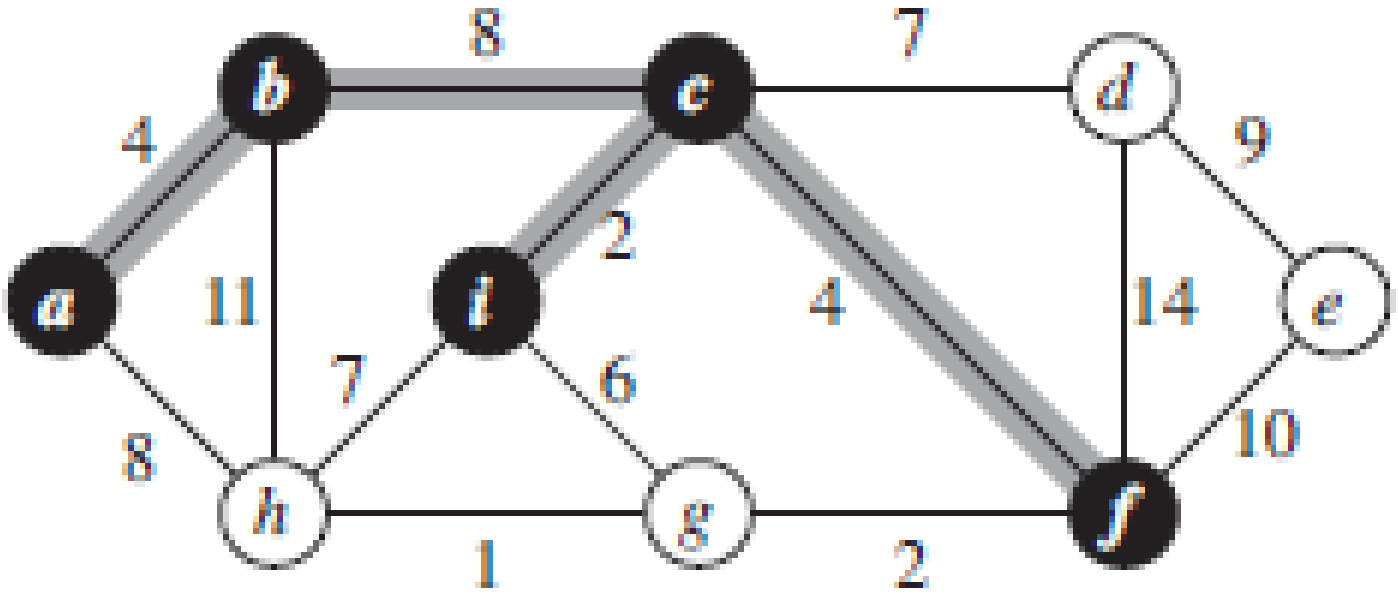


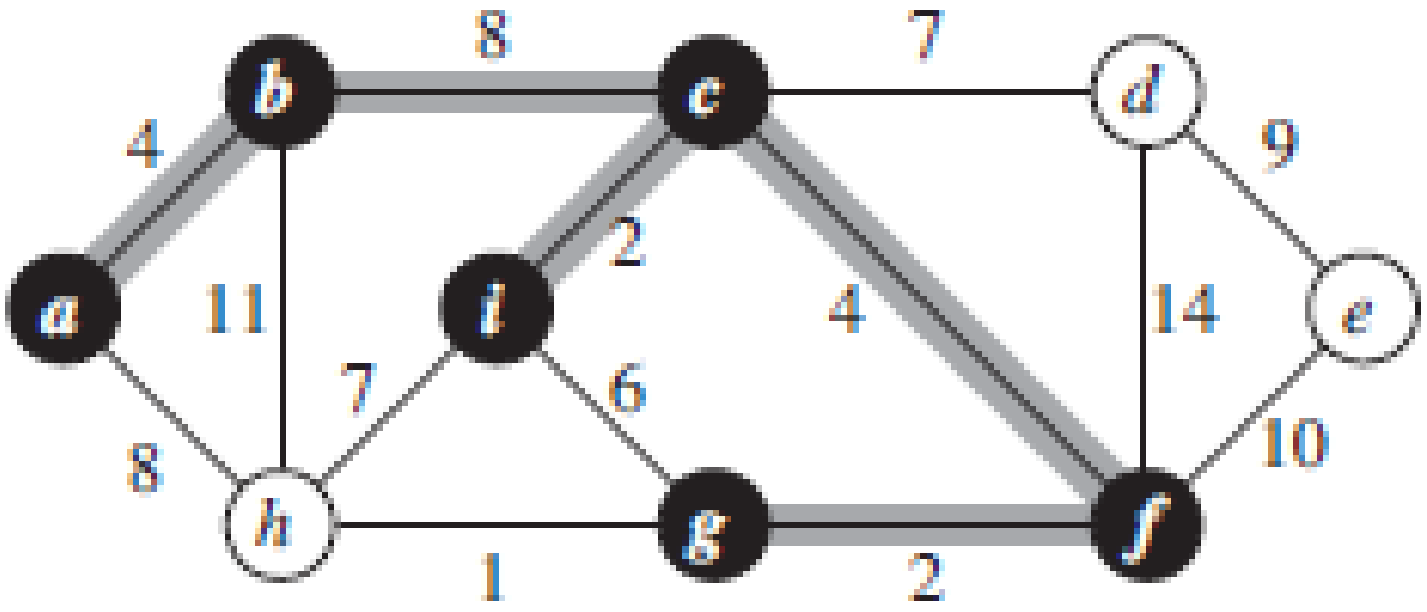

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2

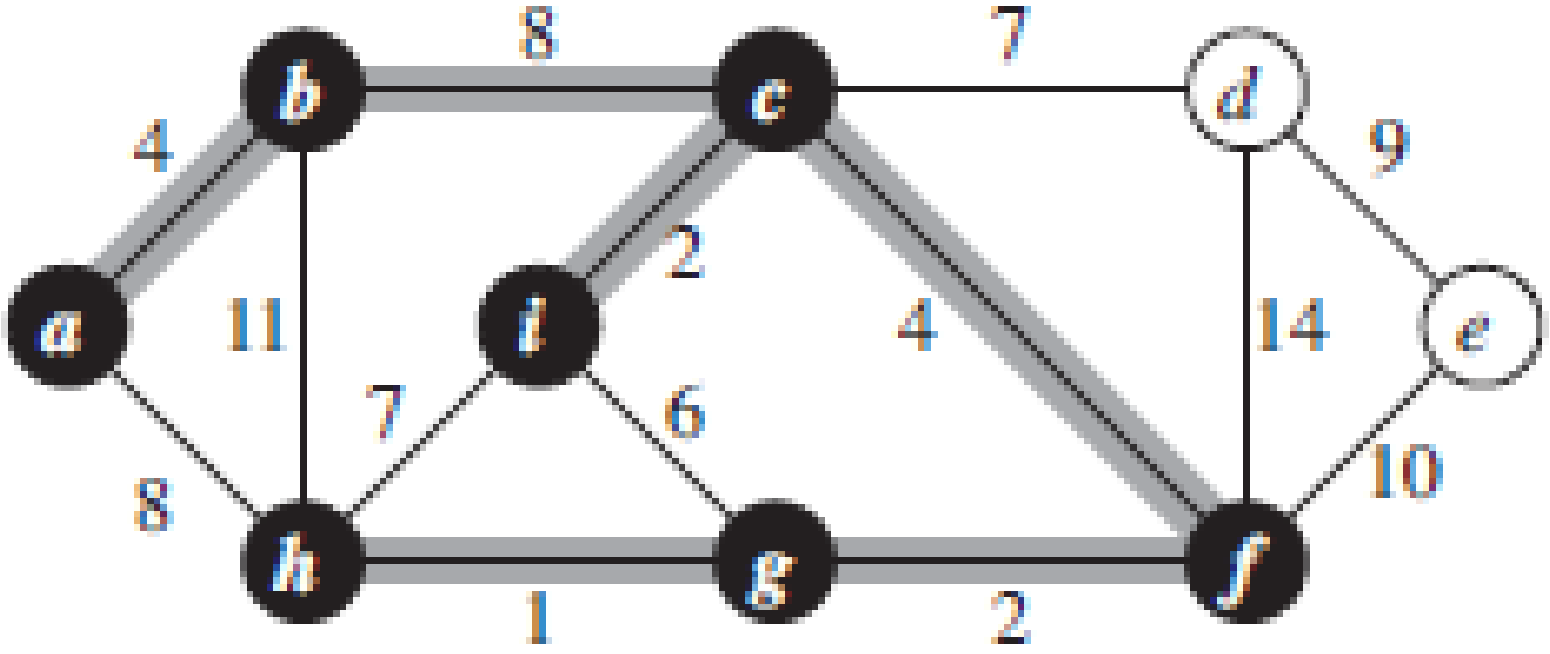

a
b
c
d
e
f
g
h
i
4
8
7
9
11
14
4
2
8
7
6
10
1
2

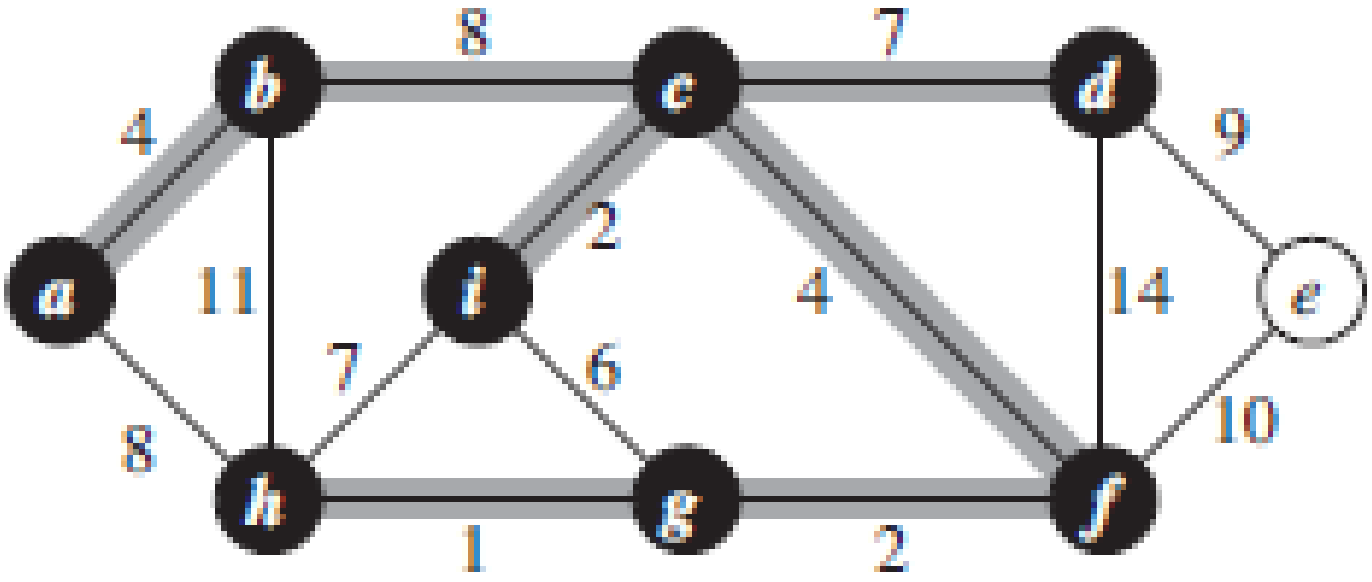

a
b
c
d
e
f
g
h
i
4
8
7
9
11
14
10
2
4
7
6
8
1
2

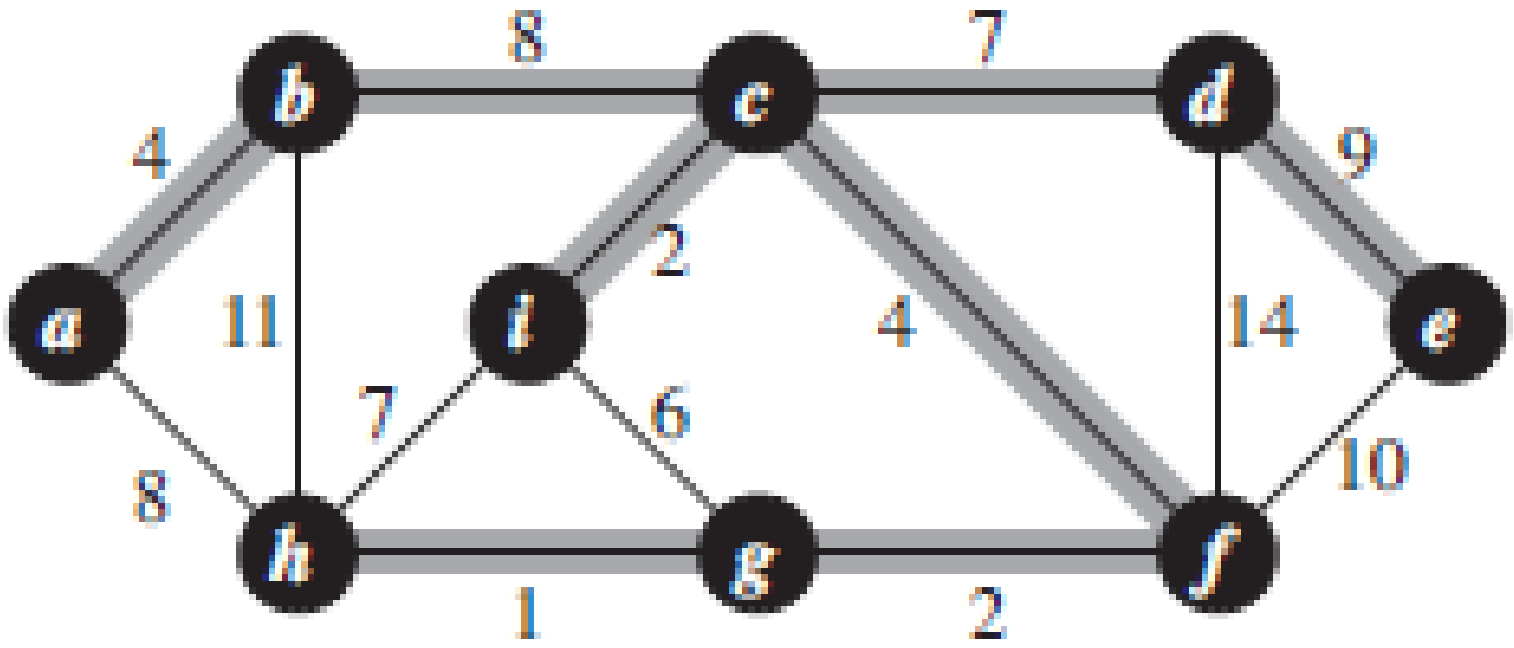

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2

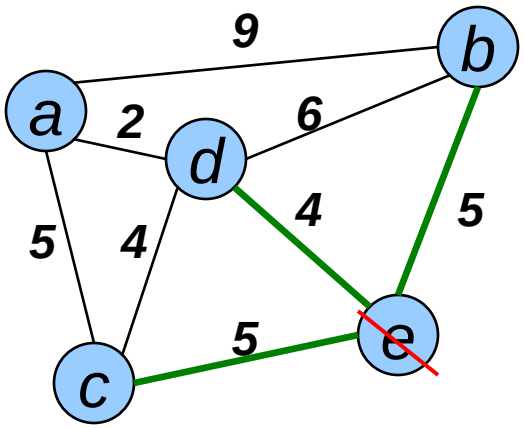

a
b
c
d
e
f
g
h
8
7
4
9
11
2
4
14
7
6
8
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
14
4
2
8
7
6
10
1
2

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Prim's algorithm



e	d	b	c	a
0	∞	∞	∞	∞

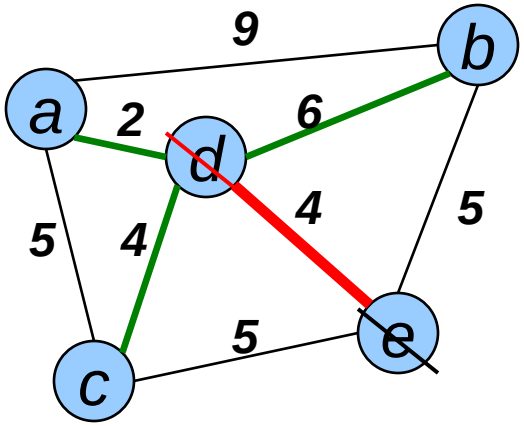
Vertex	Parent
e	-
b	-
c	-
d	-

d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

*The MST initially consists of the vertex **e,** and we update the distances and parent for its adjacent vertices*

Prim's algorithm



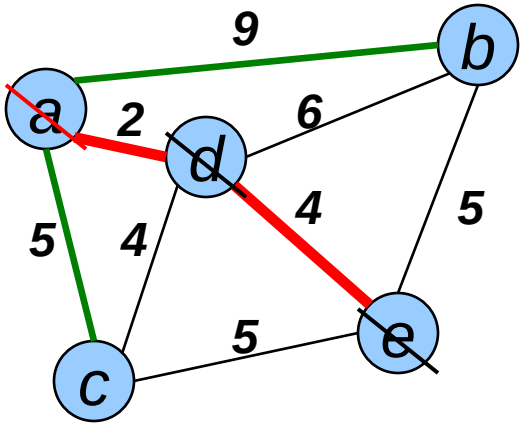
d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



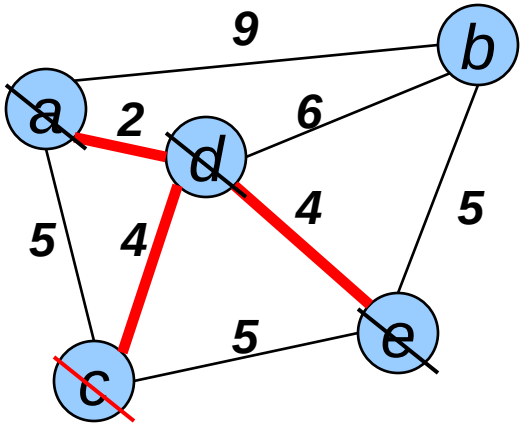
a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

c	b
4	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



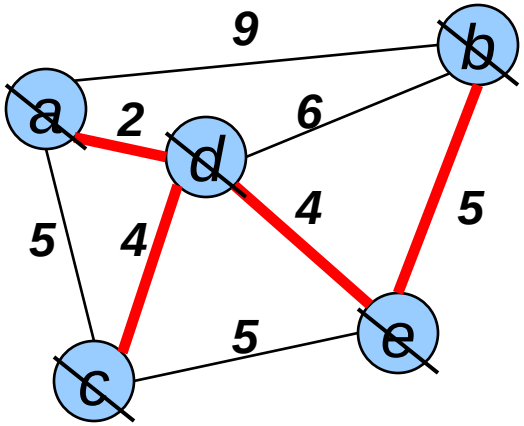
c	b
4	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Running time of Prim's algorithm (without heaps)

Initialization of priority queue *(array): O(|**V**|)*

***Update loop**: |**V**| calls*

- *Choosing vertex with minimum cost edge: O(|**V**|)*
- *Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of O(|**E**|) updates*

***Overall cost without heaps**:* $O(|E| + |V|^2)$

When heaps are used, apply same analysis as for Dijkstra's algorithm (p.469) (good exercise)

Prim's Algorithm Invariant

- At each step, we add the edge (***u,v***) s.t. the weight of (***u,v***) is **minimum** among all edges where ***u*** is in the tree and ***v*** is not in the tree
- Each step maintains a minimum spanning tree of the vertices that have been included thus far
- When all vertices have been included, we have a MST for the graph!

Correctness of Prim's

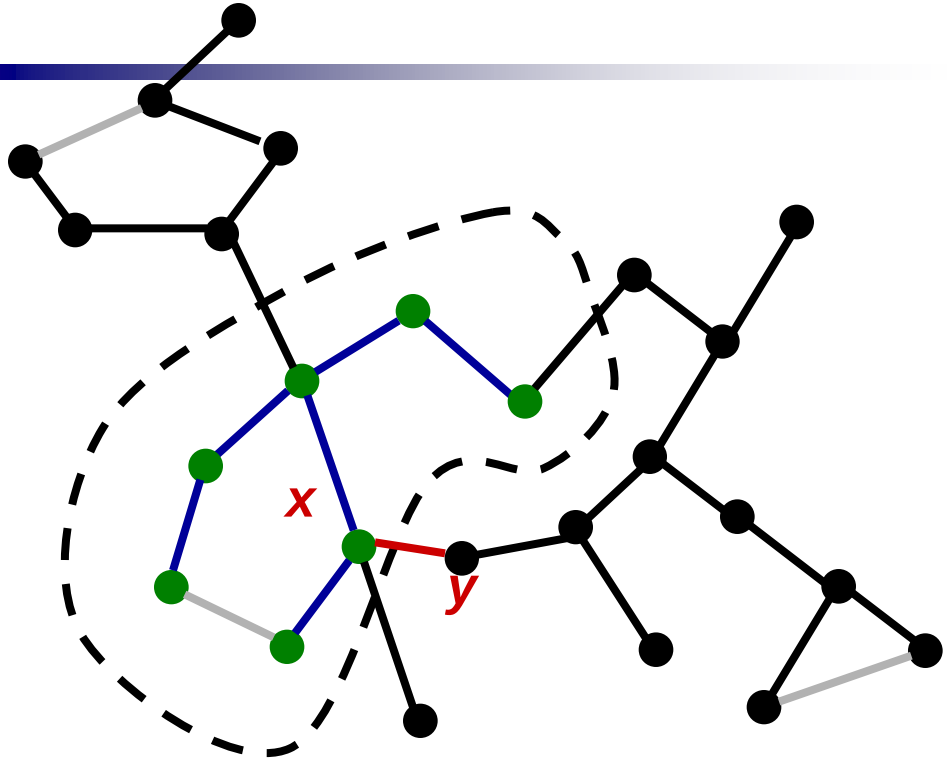
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

But is this a *minimum* spanning tree?

Suppose it wasn't.

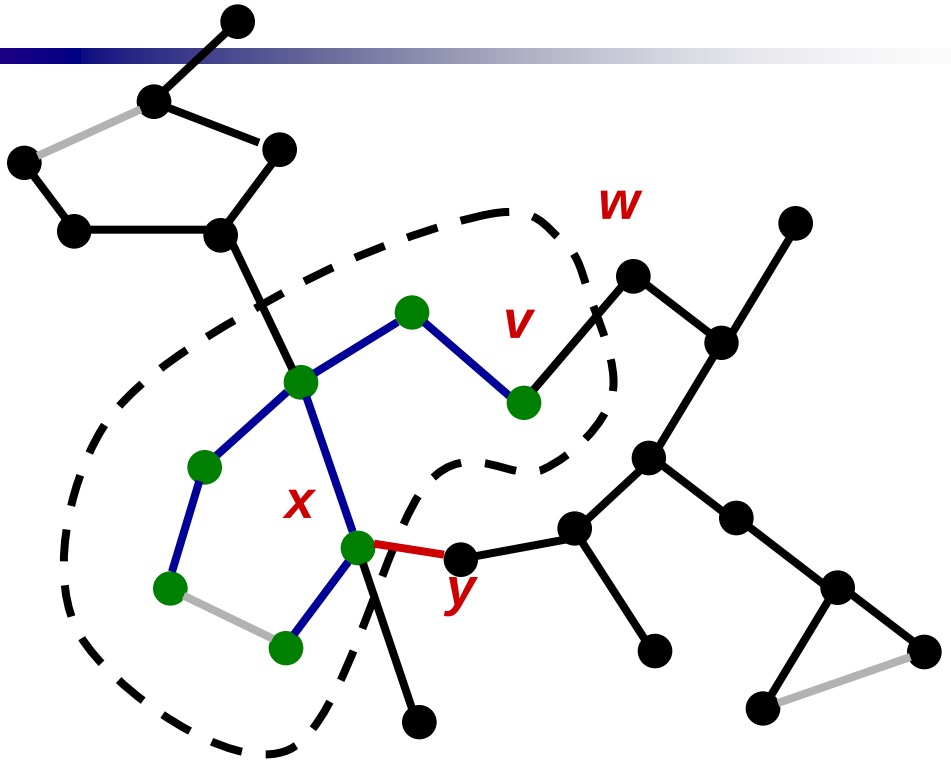
- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

Correctness of Prim's



- *Let **G** be a connected, undirected graph*
- *Let **S** be the set of edges chosen by Prim's algorithm before choosing an errorful edge **(x,y)***
- *Let **V'** be the vertices incident with edges in **S***
- *Let **T** be a MST of **G** containing all edges in **S**, but not **(x,y)**.*

Correctness of Prim's



- *Edge **(x,y)** is not in **T**, so there must be a path in **T** from **x** to **y** since **T** is connected.*
- *Inserting edge **(x,y)** into **T** will create a cycle*
- *There is exactly one edge on this cycle with exactly one vertex in **V'**, call this edge **(v,w)***

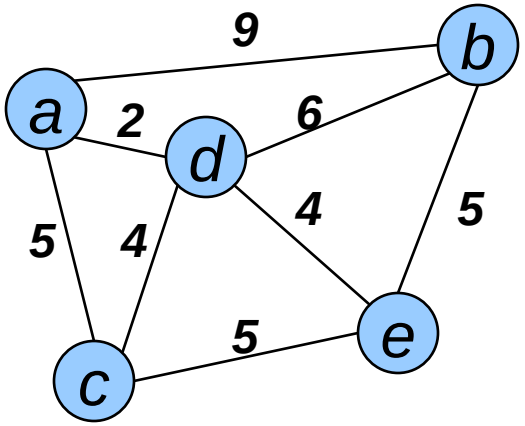
Correctness of Prim's

- Since Prim's chose **(*x*,*y*)** over **(*v*,*w*)**, w(**v**,**w**) >= w(**x**,**y**).
- We could form a new spanning tree **T'** by swapping **(*x*,*y*)** for **(*v*,*w*)** in **T** (*prove this is a spanning tree*).
- w(**T'**) is clearly no greater than w(**T**)
- But that means **T'** is a MST
- And yet it contains all the edges in **S**, and also **(*x*,*y*)**

...Contradiction

Another Approach

- *Create a forest of trees from the vertices*
- *Repeatedly merge trees by adding "**safe edges**" until only one tree remains*
- *A "safe edge" is an edge of minimum weight which does not create a cycle*



forest: *{a}, {b}, {c}, {d}, {e}*

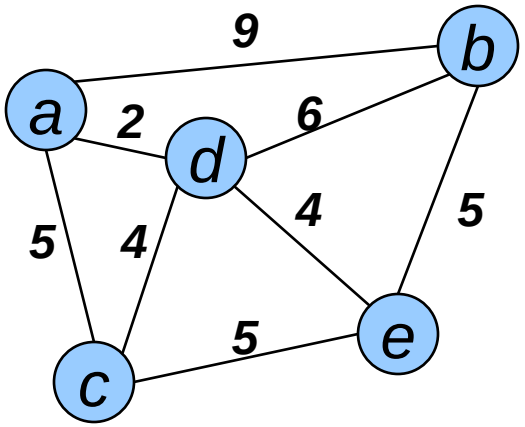
Kruskal's algorithm

Initialization

a. Create a set for each vertex $\boldsymbol{v \in V}$

b. Initialize the set of "safe edges" ***A*** *comprising the MST to the empty set*

c. Sort edges by increasing weight



$\boldsymbol{F} = \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$

$\boldsymbol{A} = \varnothing$

$\boldsymbol{E} = \{(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)\}$

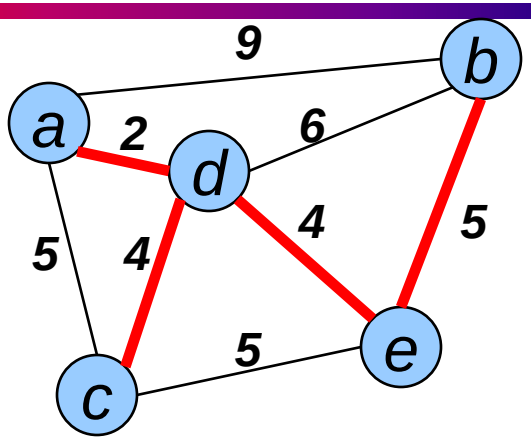
Kruskal's algorithm

For each edge (u,v) $\in$ E in increasing order
while more than one set remains:
*If **u** and **v**, belong to different sets **U** and **V***
*a. add edge (**u,v**) to the safe edge set*
A = A $\cup$ ***{(u,v)}***
*b. merge the sets **U** and **V***
F = F - U - V + $($***U*** $\cup$ ***V***$)$

Return A

- Running time bounded by sorting (or findMin)
- O(|***E***|**log**|***E***|), or equivalently, O(|***E***|**log**|***V***|) (***why???***)

Kruskal's algorithm



$E = \{(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)\}$

Forest	***A***
{a}, {b}, {c}, {d}, {e}	∅
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

Kruskal's Algorithm Invariant

- After each iteration, every tree in the forest is a MST of the vertices it connects

- Algorithm terminates when all vertices are connected into one tree

Correctness of Kruskal's

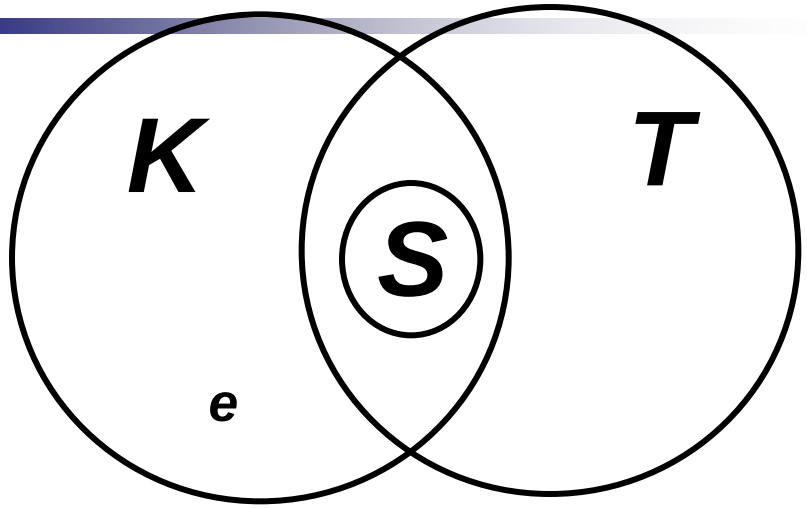
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

But is this a *minimum* spanning tree?

Suppose it wasn't.

- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

Correctness of Kruskal's



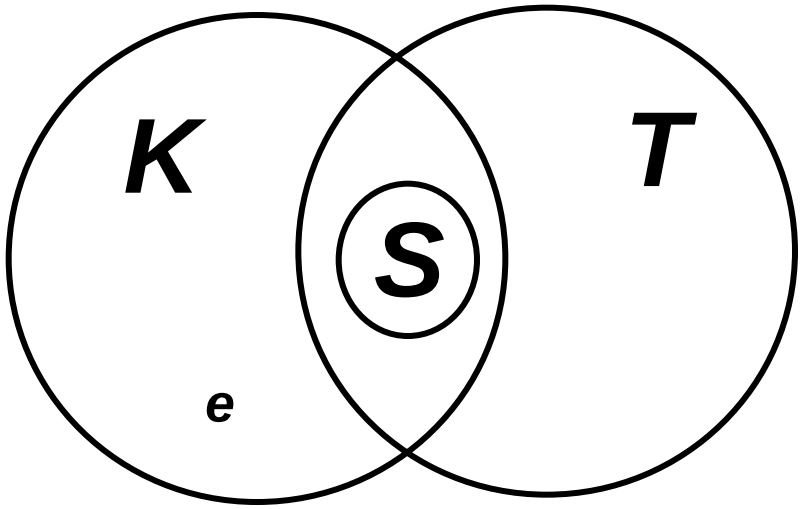
- Let **e** be this first errorful edge.
- Let **K** be the Kruskal spanning tree
- Let **S** be the set of edges chosen by Kruskal's algorithm *before* choosing **e**
- Let **T** be a MST containing all edges in **S**, but not **e**.

Correctness of Kruskal's

*Lemma: w(**e'**) >= w(**e**) for all edges **e'** in **T - S***

Proof (*by contradiction*):

- Assume there exists some edge **e'** in **T - S**, w(**e'**) < w(**e**)
- Kruskal's must have considered **e'** before **e**



- *However, since **e'** is not in **K** (why??), it must have been discarded because it caused a cycle with some of the other edges in **S**.*
- *But **e' + S** is a subgraph of **T**, which means it cannot form a cycle* ***...Contradiction***

Correctness of Kruskal's

- Inserting edge **e** into **T** will create a cycle
- There must be an edge on this cycle which is not in **K** (*why??*). Call this edge **e'**
- **e'** must be in **T - S,** so (by our lemma) w(**e'**) >= w(**e**)
- We could form a new spanning tree **T'** by swapping **e** for **e'** in **T** (*prove this is a spanning tree*).
- w(**T'**) is clearly no greater than w(**T)**
- But that means **T'** is a MST
- And yet it contains all the edges in **S**, and also **e**

...Contradiction

Greedy Approach

- Like Dijkstra's algorithm, both Prim's and Kruskal's algorithms are **greedy algorithms**

- The greedy approach works for the MST problem; however, **it does not work for many other problems**!

That's All!